Strategy Synthesis and Compression for Multi-Agent Autonomous Systems: A Correctness-Guaranteed Approach

Rong Gu[a], Peter G. Jensen[b], Cristina Seceleanu[a], Eduard Enoiu[a], Kristina Lundqvist[a]

[a] *Mälardalen University, Sweden*
[b] *Aalborg University, Denmark*

Abstract

Planning is a critical function of multi-agent autonomous systems, which includes path finding and task scheduling. Exhaustive search-based methods such as model checking and algorithmic game theory can solve simple instances of multi-agent planning. However, these methods suffer from a state-space explosion when the number of agents is large. Learning-based methods can alleviate this problem but lack a guarantee of the correctness of the results. In this paper, we introduce MoCReL, a new version of our previously proposed method that combines model checking with reinforcement learning in solving the planning problem. The approach takes advantage of reinforcement learning to synthesize path plans and task schedules for large numbers of autonomous agents, and of model checking to verify the correctness of the synthesized strategies. Further, MoCReL can compress large strategies into smaller ones that have down to 0.05% of the original sizes, while preserving their correctness, which we show in this paper. MoCReL is integrated into a new version of Uppaal Stratego that supports calling external libraries when running learning and verification of timed games models.

Keywords: Planning, Multi-Agent Autonomous Systems, Timed Games,

Email addresses: rong.gu@mdu.se (Rong Gu), pgj@cs.aau.dk (Peter G. Jensen), cristina.seceleanu@mdu.se (Cristina Seceleanu), eduard.paul.enoiu@mdu.se (Eduard Enoiu), kristina.lundqvist@mdu.se (Kristina Lundqvist)

April, 2022

1. Introduction

Autonomous agents (or shortly, agents), such as driverless cars, drones, and mobile robots, are systems that can move, carry out tasks, and collaborate with other agents autonomously without human intervention. *Multi-Agent Autonomous Systems* (MAS) [1] consist of multiple agents that work together in an environment and aim to achieve a common goal, an example being a group of construction equipment quarrying, crushing, and transporting stones. Planning for MAS involves *path finding* and *task scheduling*, and is one of the most critical problems when designing such systems [2]. There exist algorithms that solve each problem, respectively. A* [3] and rapidly-exploring random tree (RRT) [4] are two well-known algorithms that calculate the shortest paths in an environment with static obstacles. Algorithms for task scheduling have also been widely researched, resulting in search-based methods [5, 6] and learning-based methods [7, 8].

Nevertheless, approaches that solve the entire planning problem for MAS, which also provide a correctness guarantee are often not scalable [9, 10]. Learning-based methods address this weakness but fail to provide a formal guarantee of the correctness of their results. A united solution that solves both path finding and task scheduling is still missing. The difficulties of finding such a solution are threefold. *First*, the tasks of the agents are of different kinds. Some must be done individually, whereas some need collaborations, that is, agents gather at the same position and start and finish a common task simultaneously. In addition, tasks have uncertain completion time, which increases the difficulty of task scheduling dramatically. *Second*, tasks can be scheduled differently: periodically (repeatedly perform A), sequentially (perform A, then B, then C), or as a request-response pair (whenever A occurs, perform B). *Third*, the complexity of solving the problem increases exponentially when the number of agents increases linearly. This difficulty stems from the fact that task scheduling is

NP-hard [11]. Solving the problem algorithmically on MAS resulting from composing all agents' behaviors is computationally demanding.

We have previously proposed MCRL (Model Checking + Reinforcement Learning) [12, 13] as a method that combines model checking with reinforcement learning to synthesize and verify plans of agents. MCRL benefits from both model checking and reinforcement learning so that the scale of the problem that MCRL can solve is larger than that of search-based methods, and also the results (a.k.a., plans) are guaranteed to be correct by model checking. However, MCRL has some limitations: (i) models are hard to build manually when the environment is big or the agents are many; (ii) MCRL only supports simple tasks that are executed individually and periodically; (iii) the resulting plan synthesized by MCRL is larger than needed, as it contains a tabulation of system states that are unreachable under the plan, which is impeding understandability (by an operator) and the realizability on systems with limited resources.

To alleviate these issues, we propose *MoCReL* (Model-checked Compressed Reinforcement Learning), an improved and extended version of MCRL. MoCReL provides functions of synthesizing, verifying, and compressing plans, and it relies on modeling MAS as *(Stochastic) Timed Games* in Uppaal Stratego [14], which is a tool that incorporates a symbolic model checker Uppaal [15], a statistical model checker Uppaal SMC [16], a solver for *Timed Games* Uppaal TiGa [17], and solvers for *Stochastic Timed Games* relying on learning algorithms [14]. Similar to MCRL, the plan synthesis in MoCReL is an iterative process of a random simulation and reinforcement learning. The simulation explores the MAS model randomly and samples the model's execution traces that record the executed *action* at each *state* of the model. Then the learning algorithm uses these traces to synthesize a plan, which is used in the next round of simulation. This iteration ends with a final plan generated until reaching the maximum rounds of iteration, or a user-defined number of traces are sampled. Next, to guarantee the correctness of the plan, MoCReL verifies it by model checking the MAS model under the control of the plan, that is, the plan controls the model to choose certain actions at different states. The selected

pairs of state and action are labeled during the verification, which in turn helps compressing the plans. The unlabeled pairs are considered useless for satisfying the requirements, and thus are removed from the plan. In this way, plans are compressed while preserving the satisfied requirements. All the activities of plan synthesis, verification, and compression are implemented as an external library that is linked to Uppaal Stratego, which enables us to easily change or extend the algorithms for learning and compression.

In summary, MoCReL overcomes the limitations of MCRL as follows, which are the contributions of this paper:

(i) Models in MoCReL are instances of templates, which facilitates automatic model construction. In our experiments (Section 5), we design and use a tool that is capable of generating the models based on the templates.

(ii) The model templates also allow for more task types, such as collaborations among agents and tasks that are activated by events.

(iii) MoCReL's method for plan synthesis and compression is proven to be sound, that is, plans that are synthesized and compressed by MoCReL must be correct.

(iv) Experiments of MoCReL on a real industrial case study shows that the compressed plans can take down to 0.05% of the memory space of the original plans, while preserving their properties, e.g., always eventually finishing all tasks.

The remainder of the paper is organized as follows. In Section 2, we introduce the preliminaries: timed games and strategies in Uppaal Stratego, and reinforcement learning. Section 3 describes the problem of MAS planning. In Section 4, we describe our proposed methods for strategy synthesis, verification, and compression in MoCReL. Next, we present the experimental evaluation in Section 5. In Section 6, we compare to related work, and conclude the paper in Section 7, where we also mention directions for future work.

2. Preliminaries

In this section, we recall the timed automata formalism as used in the UPPAAL tool suite, timed games, and the reinforcement learning algorithm used in this paper. We denote non-negative integers as $\mathbb{N}$, and real numbers as $\mathbb{R}$.

2.1. UPPAAL Timed Automata

A *timed automaton* (TA) is finite-state automaton extended with real-valued variables [18]. The variables model the logic clocks in systems, which are zero initially and then increase synchronously with the same rate. UPPAAL [15] is a tool for modeling, simulation, and model checking of UPPAAL *timed automata* (UTA), which is an extension of TA. A UTA is defined as a tuple:

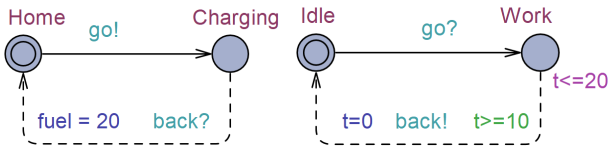
$$< L, l_0, \Sigma, V, C, E, I >, \tag{1}$$

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

The semantics of a UTA is defined as a *timed transition system* over states $q = (l,\ c)$, where l is a location, $c \in \mathbb{R}^C$ is the valuations of the clocks at this location, with the initial state $q_0 = (l_0, c_0)$, where c_0 assigns all clocks in C to zero. There are two kinds of transitions:

(i) *delay transitions*: $q_n \xrightarrow{d} q'_n$, where $n \in \mathbb{N}$, $c \models I(l)$, $q'_n = (l, c \oplus d)$ is the next state delaying from q_n, and $c \oplus d$ is obtained by incrementing all clocks with the delay amount d such that $c \oplus d \models I(l)$, and

(ii) *discrete transitions*: $q_n \xrightarrow{a} q_{n+1}$, where $q_{n+1} = (l', c')$ is the next state traversing via the edge $l \xrightarrow{g,a,r} l'$ from q_n, for which the guard g evaluates to *true* in the source state q_n, $a \in \Sigma$ is an action, and valuation of c' on the target state q_{n+1} are obtained by resetting all clocks in $r \subseteq C$ such that $c' \models I(l')$.



(a) A TG of a car (b) A TG of a charging station

Figure 1: An example of a network of TG.

A *timed game* (TG) is a TA with its set of actions partitioned into *controllable* (Σ_c) and *uncontrollable* (Σ_u) ones. Uppaal Stratego [14] is a tool that supports modeling and verifying TG as well as synthesizing strategies to solve TG. Fig. 1 depicts two templates of TG in Uppaal Stratego, which consist of *locations* and *edges*. A template may also have local declarations and parameters and can be instantiated by a process assignment (in the system definition) [15]. In a TG template, locations (e.g., `Charging`) are blue circles. The double circles (e.g., `Home`) denote the initial location. *Clocks* (e.g., `t`) are special variables that increase simultaneously at rate 1, when the TG is executed. *Invariants* (e.g., `t<=20`) on locations must be *true* when the TG stays at the location. *Edges* connecting locations denote *discrete actions*, which are partitioned into *controllable* ones (solid lines) and *uncontrollable* ones (dashed lines). *Delays* allow time to elapse on locations as long as the associated invariants are not violated. *Guards* (e.g., `t>=10`) on edges must be *true* when the edges are enabled for transition. *Assignments* on edges reset clocks (e.g., `t=0`) or update data variables (e.g., `fuel = 20`). A *network* of TG is a parallel composition of TG that can synchronize via *channels* (e.g., `go!` is synchronized with `go?`).

When TG are executed, the choices of delaying at locations or executing discrete actions are non-deterministic, whereas *Stochastic Timed Games* (STG) replace the non-deterministic choices with stochastic ones. By default, STG in Uppaal Stratego apply uniform probability distributions on discrete transitions and time-bounded delays, and exponential probability distributions on unbounded delays.

In this paper, we denote TG (STG) by $\mathcal{G}$ ($\mathcal{P}$), and the semantics of a $\mathcal{G}$ by $S_{\mathcal{G}}$. A run π of a $\mathcal{G}$ is a sequence of alternating delays (denoted by d) and discrete transitions (denoted by a) of its $S_{\mathcal{G}}$: $\pi = q_0 \xrightarrow{d_1} q'_0 \xrightarrow{a1} q_1 \xrightarrow{d_2} q'_1 \xrightarrow{a2} ... \xrightarrow{d_n} q'_{n-1} \xrightarrow{a_n} q_n$ .... If we denote the last state of a finite run π_f as $last(\pi_f)$, a *strategy* is a function that maps actions, i.e., either a controllable one $a \in \Sigma_c$ or a delay (indicated by the symbol λ), to each of the states. Formally, strategies are defined as follows [19]:

Definition 1 (Strategy)**.** *Let $\mathcal{G} =< L, l_0, \Sigma_c \cap \Sigma_u, V, C, E, I >$ be a TG. A strategy σ over $\mathcal{G}$ is a partial function: $\pi_f \rightarrow 2^{\Sigma_c \cup \{\lambda\}} \setminus \{\emptyset\}$ such that for any finite run π_f ending in state $q_l = last(\pi_f)$, if $a \in \sigma(\pi_f) \cap \Sigma_c$, then there must exist a transition $q_l \xrightarrow{a} q_{l+1} \in S_{\mathcal{G}}$.* □

A *stochastic* strategy of an STG delivers probabilities instead of definite choices of actions [19]. If we denote the set of runs in $S_{\mathcal{G}}$ as $\Pi_{\mathcal{G}}$, a TG under the control of a strategy σ as $\mathcal{G} \mid \sigma$, the outcome of running $\mathcal{G} \mid \sigma$ is a subset of $\Pi_{\mathcal{G}}$, denoted as $Out(\mathcal{G} \mid \sigma)$. $Out(\mathcal{G} \mid \sigma)$ can be defined inductively as follows[1]:

Definition 2 (Outcome of $\mathcal{G} \mid \sigma$)**.** *Given $q_0 \in Out(\mathcal{G} \mid \sigma)$, if $\pi \in Out(\mathcal{G} \mid \sigma)$ then $\pi' = last(\pi) \xrightarrow{e} q$ and $\pi' \in Out(\mathcal{G} \mid \sigma)$ if either one of the following conditions hold:*

1. *$e \in \Sigma_u$, or*
2. *$e \in \Sigma_c$ and $e \in \sigma(last(\pi))$, or*
3. *$e \in [0, T] \subseteq \mathbb{R}_{\geq 0}$ and $\forall e' < e$, $last(\pi) \xrightarrow{e'} q'$ for some q' s.t. $\sigma(q') \ni \lambda$, where T is the invariant boundary on the location of $last(\pi)$.* □

We will use these three conditions in the proof of Theorem 1. Let P be a proposition and the reachability objective for $\mathcal{G}$, a finite run π_f is *winning* w.r.t. P, if P is true at the last state of π_f. A strategy σ over a $\mathcal{G}$ is winning if all runs in $Out(G|\sigma)$ are winning. A *memoryless* strategy makes decisions on actions depending on the current state only, that is, a function σ: $last(\pi_f) \rightarrow 2^{\Sigma_c \cup \{\lambda\}} \setminus \{\emptyset\}$. In this paper, we aim to synthesize *winning*, *memoryless*, and *non-lazy* strategies, that is, winning strategies that urgently decide on a controllable

[1] Definition 2 is adapted from the definition strategy outcome in the literature [19].

action or *wait* until the environment makes a move[2]. Strategies referred to in the rest of this paper are all *memoryless* and *non-lazy*.

2.3. Model Checking and Temporal Properties

Model checking [20] traverses the state space of a formal model (e.g., UTA) and checks if it satisfies certain properties. The properties in this paper are of the following forms, where p is an atomic proposition over the locations, clocks, and data variables of the UTA:

(i) **Invariance**: $E[]\ p$ meaning that there exists a run where all the states satisfy p, or $A[]\ p$ meaning that for all runs, p is satisfied by all states in each run,

(ii) **Liveness**: $A<>\ p$ meaning that for all runs, p is satisfied by at least one state in each run.

2.4. Reinforcement Learning

Reinforcement learning (RL) [21] is a kind of machine learning method for training agents by assigning rewards to desired behaviors and/or penalties to undesired ones, with the purpose of maximizing the accumulated rewards. Model-free RL relies on samples from the environment, which can be a virtual or a real one, to estimate the rewards of the future state-action pairs following the agent's current state. Model-based RL uses the model's predictions or distributions of state-action pairs and their rewards to find optimal actions. Therefore, models in the model-based RL must contain the full information of the environment and agents, which is hardly to obtain in an unexplored or partially observed environment.

Q-learning [22] is one of the model-free algorithms, which, at the limit, converges to the *optimal policy* for agents. Policies are associated with a state-action value function called *Q function*. The optimal Q function satisfies the Bellman optimality equation:

[2]Memoryless and non-lazy strategies are shown to suffice for optimal scheduling of Duration Probabilistic Automata [5].

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

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. The rewards of the pairs are often stored in a score table. We show an example of such score tables in Section 3.2.

3. Problem Description

In this section, we introduce the planning problem of MAS and its challenges.

3.1. Overall Description

MAS are designed to move and execute a series of tasks autonomously. The actions belonging to a MAS can be categorized as: (i) movement, and (ii) executing a task. Whenever an agent moves or starts a task, the environment decides the ending time of the action. Now, the MAS planning is to order these two kinds of actions such that the MAS can finish its tasks while satisfying certain requirements, e.g., never let two agents execute the same task simultaneously, no matter how the environment reacts. The overall goal of MAS planning is formulated as below:

Overall Goal: Given a MAS and a set of requirements, the goal of planning is to order the agents' actions of movement and task execution, according to their variable ending time and occurrences of events, which are decided by the environment, such that the MAS can finish its tasks and satisfy the requirements.

Remark 1. *One can reduce the planning problem to a path-finding problem by removing the actions of task execution, or a task-scheduling problem by removing*

the actions of movement. Our algorithm is capable of solving the general problem that contains path finding and task scheduling or only one of them.

Remark 2. *The requirements can be functional ones, such as always start task A after task B finishes, and safety ones, such as no collision with the static obstacles in the environment happens.*

3.2. Challenges of Solving the Planning Problem

The major challenges of this problem stem from four aspects, which get amplified especially when solving the problem via algorithmic techniques:

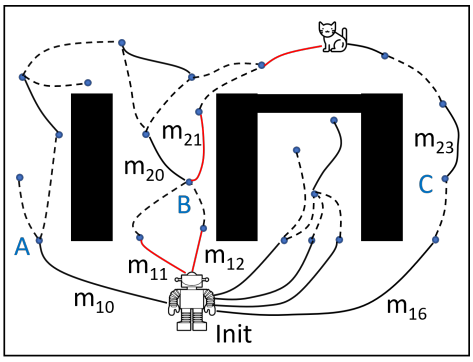
- **Challenge I** (uncertainty): The agents' actions have uncertain execution time, which means agents can choose actions to perform but cannot control how long time the actions will take. The uncertainty of execution time makes static plans inefficient, since they assign starting time to the actions without knowing their actual ending time.

- **Challenge II** (variety of task constraints): Some tasks have additional constraints, e.g., task A should always be completed before task B starts. Some tasks must be executed whenever certain events occur.

- **Challenge III** (complexity): As an NP-hard problem [11], when synthesizing and verifying plans for MAS, the state space of the model grows exponentially when the number of agents increases linearly as shown in the literature [9, 10].

- **Challenge IV** (large plans): As the state space of the problem grows exponentially, the resulting plan can grow exponentially too. However, some of the information in the plans may never be used. It is time-consuming to look for the right actions in a large plan. In some applications, it is simply impossible to store plans that take too much memory space, such as in Airborne Collision Avoidance System X case [23].

To give a concrete example of large plans, in Fig. 2, we show a path-finding problem in a 2D space, where a robot tries to catch a cat. Note that our



(a) An example of a plan for a path-finding problem. Solid lines are controllable actions in the plan. Dashed lines are the uncontrollable actions of the environment. Red lines are the ones that guarantee to reach the destination no matter how the environment reacts. Black lines are useless ones that should be removed from the plan.

	m_{10}	m_{11}	m_{12}	...	m_{16}
Init	-2	3	3	...	1
...					
	m_{20}	m_{21}			
B	1	4			
	m_{23}				
C	2				
...					

(b) The score table of the plan for this example. The first column indicates states. The grey rows indicate the controllable actions at states. The white rows show the scores of state-action pairs accumulated by reinforcement learning.

Figure 2: An example of path finding in an environment with uncertain behaviors and the score table of the path plan.

mission-planning problem combines path finding and task scheduling, which makes the model state space to be high dimensional rather than a 2D space.

Algorithmic planning methods, such as the Dijkstra's algorithm for path finding [24], and the symbolic on-the-fly algorithm for solving timed games [17], usually explore the model's state space in a certain order (e.g., depth-first exploration), store the preceding states of each state, and back propagate to the initial state when finding the goal state. The resulting plan is *concise* as it only contains the state-action pairs that are *correct*, that is, they satisfy the requirements and reach the goal state. Additionally, the correctness of the plan is guaranteed as the algorithms explore the state space exhaustively [17]. However, the algorithmic methods are not scalable and when the model's state space becomes large, they fail to solve the problem in a reasonable time [13].

A path-finding algorithm that uses reinforcement learning can alleviate this problem by replacing the exhaustive state-space exploration with random simulation [13], while in turn suffering from disadvantages that we emphasize in the following. As depicted in Fig. 2(a), a path plan synthesized by reinforcement

learning possibly explores multiple routes from the robot to the cat, and results in a score table shown in Fig. 2(b). The score table only contains controllable actions. A robot under the control of a plan always chooses the actions with the highest score at each of its states. For example, initially, the robot non-deterministically chooses one action among $\mathtt{m}_{11}$ and $\mathtt{m}_{12}$, because they have the highest score at the state `Init`. These scores are accumulated gradually during the course of learning, hence, although the score of the pair (`Init`, $\mathtt{m}_{10}$) ends to be -2, one cannot neglect it before the learning finishes. Another example of useless data is the pair (`C`, $\mathtt{m}_{23}$). It is sampled during the random simulation, but not used in the final plan, which initially chooses to do actions $\mathtt{m}_{11}$ and $\mathtt{m}_{12}$, and thus never gets to state `C`. Besides, as the synthesis is based on random simulations, there is no guarantee on the correctness of the results, that is, the actions with the highest score are not guaranteed to lead the agents towards the goal state and satisfy all requirements. Hence, there is a need of removing the useless data from the plan while guaranteeing the correctness of the result.

Overall challenge. In a nutshell, the overall challenge of MAS planning is to design a method for plan synthesis that can cope with the uncertain execution time of actions, variety of task constraints, and large state spaces of the MAS models in real cases, and for compressing large plans that could contain useless data. The compressed plans must have a correctness guarantee.

3.3. A Motivating Example

In this section, we introduce the *autonomous quarry* that serves as the industrial case-study provided by Volvo Construction Equipment (CE) in Sweden. As depicted in Fig. 3, the quarry contains various autonomous agents, e.g., trucks and wheel loaders. The goal 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. Trucks can choose to get loaded from the wheel loaders or primary crushers. After being loaded, the trucks carry the stones to a secondary crusher, which is the destination of the stones.

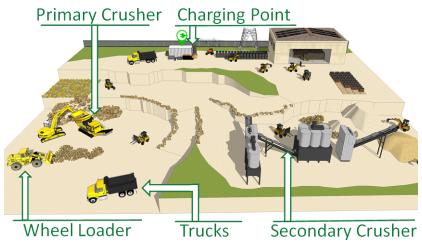


Figure 3: An autonomous quarry

During the transportation, the agents move, collaborate or work independently, and charge timely in order to achieve their goal, while satisfying requirements such as quarrying $2000m^3$ of stones per day. The challenges of the use case are as follows, which fall into the general challenges in planning problems of MAS (see Section 3.2):

- Task durations are uncertain because of the uncertainties in the environment. For instance, when trucks are unloading stones into a primary crusher, the speed of the conveyor belt on the primary crusher varies, which results in different execution times of unloading. Other trucks may need to wait until the previous one finishes its work at the primary crusher, which can even influence the entire plan (**Challenge I**).

- Some tasks are executed independently by agents, such as unloading to secondary crushers. Some tasks require collaboration between agents, such as wheel loaders loading stones into trucks. Some tasks must be prioritized when certain events occur, such as the charging task that must be prioritized when the agent's battery/fuel level is low (**Challenge II**).

- According to the experience of Volvo CE, the number of agents can vary from 2 to 8. However, our previous study has demonstrated that synthesizing correctness-guaranteed plans by using model checking is limited to MAS with less than 5 agents [9]. Handling larger numbers of agents is challenging (**Challenge III** and **Challenge IV**).

To overcome these challenges of MAS planning, we design an approach called MoCReL, which is an improved version of MCRL that we have proposed previously [12]. MCRL combines model checking with reinforcement learning, so it can deal with more agents than the algorithmic methods do, however, its task types do not support collaborations and events in MCRL, and large plans can

not be compressed either. Next, we introduce MoCReL in detail.

4. Strategy Synthesis, Verification and Compression

In this section, we introduce the workflow of MoCReL and describe the TG of MAS together with the important techniques that are used in MoCReL for strategy synthesis, verification, and compression.

4.1. Overall Workflow of MoCReL

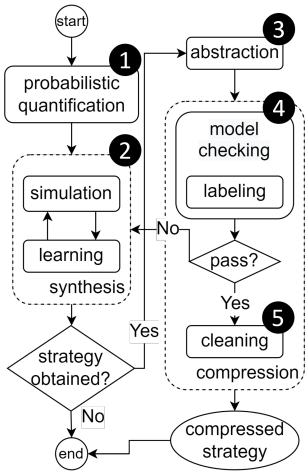


Figure 4: Workflow of MoCReL

The workflow of MoCReL is shown in Fig. 4.

Step 1: A probabilistic quantification is conducted on the TG to facilitate sampling over the system, effectively turning the TG into an STG (Stochastic Timed Game).

Step 2: Strategy synthesis takes place, which employs the Monte-Carlo simulation in Uppaal Stratego [14] to simulate the models and sample runs that satisfy certain properties. Next, the sampled runs are passed to the reinforcement learning module to generate strategies. Iterations between the simulation and learning continue until reaching the limit of iteration or sampling a user-defined number of runs. In this paper, we extend Uppaal Stratego such that it supports using external libraries to change the learning module [25], and implement MoCReL as an external library[3].

Step 3: When the synthesis finishes, a stochastic strategy is obtained, which is then abstracted as a non-deterministic strategy and verified.

Step 4: During the verification, the model checker inquires the external library, where the strategy is stored, about the allowed/preferred actions at a given state. The preferred state-action pairs are labeled as "visited".

[3]The introduction and an example of the library are in Appendix A.4.

Step 5: If the verification fails, we go back to the *Step 2* with an increased number of iteration limit so that the new round of synthesis can have more samples for learning. If the verification passes, the unlabeled pairs are removed from the strategy so that the compressed strategy takes less memory space.

Models and strategies throughout the workflow are interpreted semantically as shown later in Subsection 4.4. Uppaal Stratego supports both the algorithmic synthesis in Uppaal TiGa [26] and the learning-based synthesis that uses reinforcement learning [19]. Results of the algorithmic synthesis are correct-by-construction, but the method does not scale as it needs to explore the state spaces of the models exhaustively. In MoCReL, we propose a post-verification of the strategies that are synthesized by learning. The verification is exhaustive so the results are guaranteed to be correct. Moreover, as the verification is conducted on the agent model controlled by a strategy, the state space can be much reduced. Therefore, problems that are too complex to be handled by Uppaal TiGa can be solved by MoCReL.

4.2. Modeling of MAS

MoCReL models the agent behaviors into timed games (TG), including: (i) movement TG that model the connection and traveling time between every pair of legal positions in the environment. Legal positions are the ones that are accessible by the agents; (ii) task execution TG that model the switch between tasks and the idle state, and the task execution time; (iii) monitor TG that monitor events. When an event occurs, a monitor TG informs task execution TG to execute the corresponding task.

As a major difference between MoCReL and our previous approach MCRL [12], the models in MoCReL are much easier to adapt to different scenarios of the planning problem, as they are instantiated from the model templates. One does not need to change the templates but only instantiate the templates with different values of parameters in order to fit in one's own application. The figures in this section only illustrate the brief structures of the model templates, without showing the complex guards and functions on the edges. The full templates are

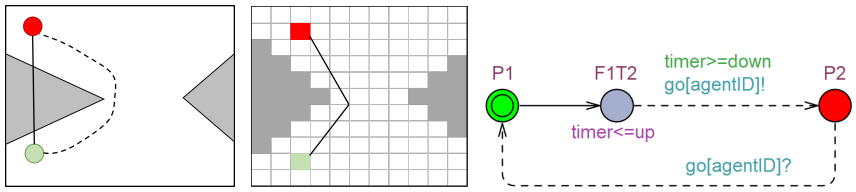


Figure 5: Examples of trajectories and the TG template of agent movement.

shown in Appendix A.3.

(i) **Movement TG**: The TG template of movement models agent traveling from one point to another. The points can be anywhere except the obstacles within the map. Since the purpose of the model is to synthesize plans, the movement TG do not model the concrete trajectories of the agents, but the traveling directions and times. Fig. 5(c) shows the brief structure of the template of *movement TG*, in which locations `P1` and `P2` represent any legal positions in the map. The location `F1T2` models the duration of traveling from `P1` to `P2`. Although the edge from `F1T2` to `P2` is uncontrollable by agents, the invariant (`timer` $\leq$ `up`) and guard (`timer` $\geq$ `down`) regulate that the traveling time must be within the interval between `down` and `up`. As one template only models one traveling direction, when the agent travels in the opposite direction, i.e., from `P2` to `P1`, the traveling time is counted by another TG that models the converse direction of movement. This TG (`P1` to `P2`) is synchronized with the other TG (`P2` to `P1`) on the channel `go` with the index of the agent.

Fig. 5 shows different modeling granularity of traveling from the green position to the red position. Even though there exists an obstacle between the two positions, one can model the movement as one instance of the movement TG template, which only reflects the existence of a movement between these two positions (the solid line in Fig. 5(a)), and the traveling time. In this case, solving the mission-planning problem aims at computing a high-level plan that does not concern how the agents maneuver in order to carry out the movement safely (the dashed line in Fig. 5(a)). Alternatively, one can model the move-

ment as several instances of the template, which reflect the discrete segments of the trajectory (Fig. 5(b)). When using MoCReL to purely solve a path-finding problem, one can model the movement from one unit of the discrete map to another as an instance of the template. The granularity of modeling depends on the users' applications.

(ii) **Task execution TG**: Similar to the movement TG, the task execution TG do not model the concrete steps of executing a task, but only the switch between task execution and idle, and the execution time of the task. There are several different templates designed for different types of tasks, such as tasks without precondition, tasks with events, and tasks that need agents to collaborate. One can instantiate the templates according to one's own application by assigning values to the parameters of the templates, such as BCET and WCET (best-case and worst-case execution time, respectively), preconditions, and the event that activates the task. However, the structure of the templates is the same (Fig. 6).

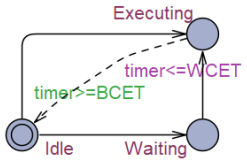


Figure 6: TG template of task execution

When the agent is allowed to execute a task, the edge from location `Idle` to `Executing` is enabled. A task being allowed means the following conditions are *true*: the task has not been finished yet, its precondition such as some certain tasks having been finished is *true*, the agent is at the right position where the task is allowed to run, and the events that activate the special tasks have not occurred (respectively, occurred) if the current one is a regular (respectively, special) task. Additionally, if a task needs a collaboration among agents, the collaborating agents must be ready and located at the same position.

When the task is ready but the device that is required by this task is taken by another agent, the agent can choose to wait, i.e., transfer to location `Waiting`, and change to location `Executing` when the device is free. When the task is being executed, the TG *can* leave location `Executing` when the timer exceeds the BCET, and *must* leave the location when it exceeds the WCET, meaning that the execution time of the task is between BCET and WCET. One thing that is worth mentioning is that agents are allowed to move only when not executing

any task, i.e., the task execution TG is at the *Idle* location. We use a global Boolean variable shared by the movement TG and the task execution TG to indicate whether the agent is idle or not.

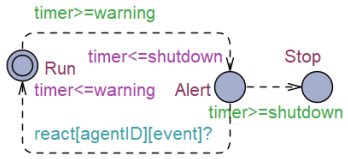


Figure 7: TG template of monitors

(iii) **Monitor TG**: A monitor monitors a signal and triggers an event when the signal exceeds a certain threshold. For simplicity, we assume the signals to be changing monotonically with time. Since the tool that MoCReL relies on, i.e., Uppaal Stratego, allows defining ordinary differential equations (ODE) of continuous variables, one can eliminate this assumption by assigning ODE to locations. However, we leave this for the future work.

Based on the assumption, a monitor TG watches the elapse of time instead of the signal, and triggers an event when time elapses a certain period, meaning that the signal exceeds a certain threshold. In Fig. 7, when the `timer` exceeds a particular constant integer (i.e., `warning`), the monitor TG transfers to location `Alert` while updating a variable representing the event. The corresponding task execution TG (Fig. 6) is then activated in the sense that its edge for starting the task is enabled. If the agent can finish the task before the `timer` reaches the limit, which is represented by a constant integer (i.e., `shutdown`), the monitor TG moves back to the initial location to restart the monitoring; otherwise, the monitor goes to location `Stop`, when all controllable actions of the agent are not allowed to be taken any more, meaning that the agent stops operating.

We call the network of movement TG, task execution TG, and monitor TG a MAS TG. Properties of a MAS TG can be expressed by a subset of Computation Tree Logic (CTL) [15] that is supported by Uppaal Stratego. Since the formal models of MAS have been defined, we can now define the planning problem formally before introducing the approach in detail.

Definition 3 (Planning)**.** *Given a MAS TG $\mathcal{G}$ and a liveness property $\mathcal{Q}$ in the form of A<> p, the planning problem $\mathcal{M} =< \mathcal{G}, \mathcal{Q} >$ reduces to generate a strategy σ over $\mathcal{G}$ such that $\mathcal{G}$ can satisfy $\mathcal{Q}$ when it is controlled by σ, i.e.,*

$\mathcal{G} \mid \sigma \models \mathcal{Q}$. □

The liveness property $A<> p$ means that $\mathcal{G} \mid \sigma$ will always eventually satisfy p. As the main goal of mission planning is to find the strategy that controls the agents to finish all their tasks eventually no matter how the environment reacts, the liveness property is used in the synthesis. The correctness guarantee of other requirements, such as safety, can be achieved by the verification after a plan is synthesized. We will give more details on these properties in Section 4.4.

4.3. Partial State-Space Observation

During the learning iteration, numerical rewards of taking an action at a state are used by reinforcement learning (e.g., the Bellman equation in Q learning [22]) to populate a score table of state-action pairs. When the learning finishes, the final values of the pairs are stored in the score table, which serves as a strategy. Before introducing how the strategy is used, in this subsection, we introduce another important concept in MoCReL: *partial observation* of the state space.

The learning algorithms need to identify the states of MAS to build up the score tables. As a formal model, MAS TG provides a clear definition of states, consisting of locations, clock values, and other data variables (Section 2). However, the strategies of MAS TG do not necessarily need to know all the components of states. For example, if the planning problem does not contain timing properties, e.g., finishing the tasks within 1 hour, the strategies can ignore all the clocks of the states, which simplifies the problem by eliminating unnecessary details. Hence, we use a partial observation of the state space of a MAS TG, which is supported by Uppaal Stratego. One can simply provide the interesting variables of the MAS TG to the learning algorithm so that the synthesized strategies do not contain unnecessary information. Details of specifying partial observability is given in Query (3) in Subsection 4.4.

4.4. Key Techniques of MoCReL

In this section, we will give a detailed introduction of the key techniques used in MoCReL after the definition of strategies that we synthesize in this paper.

4.4.1. Strategy Definition

What MoCReL aims to synthesize is a subset of *memoryless* and *non-lazy* strategies that do not contain clocks. This restriction enables us to develop an algorithm for exhaustively verifying TG under the control of learned strategies in Uppaal Stratego, which used to support exhaustive verification only on strategies with symbolic states. The trade-off of the restriction is that the planning problem in this paper does not consider timing properties. Formally, we define the strategies that MoCReL synthesizes as follows:

Definition 4 ((Stochastic) Strategy with a Score Table)**.** *Given $\mathcal{M} =< \mathcal{G}, \mathcal{Q} >$ as a planning problem of MAS, a (stochastic) strategy of $\mathcal{G}$ is a function $\sigma : q \rightarrow \mathcal{A} \subseteq \mathcal{A}^q_\mathcal{G}$ with a score table of state-action pairs, where q is a state consisting of discrete variables, and $\mathcal{A}^q_\mathcal{G} \subseteq 2^{\Sigma_c \times \{\lambda\}}$ is a set of actions that are allowed by $\mathcal{G}$ at state q. The strategy σ is considered to solve $\mathcal{M}$ if the following conditions hold, where $\|\mathcal{A}\|$ is the cardinality of $\mathcal{A}$, $max(\mathcal{A}^q_\mathcal{G})$ returns the actions with the highest score in $\mathcal{A}^q_\mathcal{G}$:*

1. *if $\|\mathcal{A}\| = 0$ (i.e., σ does not contain q), then $\forall a \in \mathcal{A}^q_\mathcal{G}$, $a \in max(\mathcal{A}^q_\mathcal{G})$;*
2. *if $\|\mathcal{A}\| \geq 1$, then $\forall a \in \mathcal{A}$, $a \in max(\mathcal{A}^q_\mathcal{G})$;*

When $\|\mathcal{A}\| \neq 1$, ties among actions happen. Non-deterministic (respectively, stochastic) strategies break the ties by non-deterministic (respectively, uniformly-distributed) choices over $\mathcal{A}$ when $\|\mathcal{A}\| > 1$, or $\mathcal{A}^q_\mathcal{G}$ when $\|\mathcal{A}\| = 0$. □

Unlike the strategies synthesized by algorithmic methods (e.g., Uppaal TiGa), the ones defined in Definition 4 do not guarantee to solve the MAS planning problem. Possible errors can exist in the design of the reward functions of the reinforcement learning algorithm, which do not reflect the desired properties in the planning problem, or the learning phase is not sufficient to

populate a score table that covers enough states. We will give some examples of the design errors in Section 4.4.3 after the query for synthesis is introduced.

In the next section, we show how MoCReL synthesizes, verifies, and compresses strategies defined in Definition 4.

4.4.2. Probabilistic Quantification and Abstraction

Due to the inherent difference between the phases of synthesis and verification, models are interpreted semantically differently in MoCReL when being simulated from when they are being verified. This is automatically done by *probabilistic quantification* and *abstraction* of the models and strategies in MoCReL.

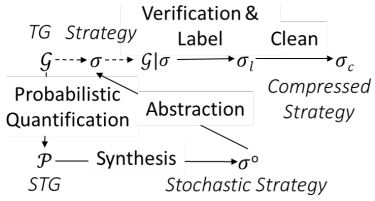


Figure 8: Model relations

Fig. 8 shows the transformation of the model semantics in the workflow of MoCReL. Initially, the MAS TG is interpreted as an STG during strategy synthesis because random simulation is needed in this step. An operation called *probabilistic quantification* changes the non-deterministic choices of actions to stochastic ones with concrete probability distributions. Specifically, time-bounded delays and discrete actions are transformed into stochastic ones with uniform distributions of probabilities. For example, in Fig. 6, the non-deterministic choice of when to leave location `Executing` is transformed to an uniformly-distributed one.

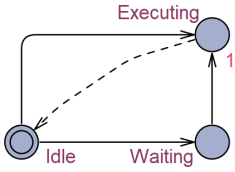


Figure 9: A task execution STG of a task with an unbounded execution time. This model does not exist in our MAS STG.

Exponential probability distributions are assigned to unbounded delays on locations with only uncontrollable actions as its outgoing edges, such as location `Executing` in Fig. 9. A constant integer of the exponential rate must be assigned to such locations (e.g., `1`). If a location has no invariant and only controllable actions as outgoing edges, such as location `Idle` in Fig. 9, the length of delay is not considered because we focus on non-lazy strategies, where agents urgently decide on a discrete action when it is available, or delay until the en-

vironment reacts. In this case, the discrete actions and delay are equally likely to be chosen initially, so the model templates do not need to be changed either.

In this paper, the duration of movement and task execution are all time bounded, so no exponential distribution exists in our MAS STG. Hence, the model templates of movement and task execution do not need to be changed, as the probabilistic quantification is done on the semantic level automatically by UPPAAL STRATEGO.

Next, synthesis based on the MAS STG generates stochastic strategies. Specifically, the simulation samples runs of the model and sends them to the learning algorithm to accumulate the scores of state-action pairs of the runs. During the learning phase, the probabilities of actions are not always the same. Actions with higher scores become more likely to be chosen than the ones with lower scores. Unexplored state-action pairs are equally likely to be chosen as the ones with the highest scores. This arrangement is referred to as "exploration" in reinforcement learning literature. After a user-defined number of runs is consumed by the learning algorithm, a stochastic strategy is considered to be generated.

After the synthesis, strategies are to be verified and compressed. To achieve verification, stochastic strategies must be transformed into non-deterministic strategies so that they can be exhaustively model checked. This step is called *abstraction* (see Fig. 8), which is also automatically carried out by UPPAAL STRATEGO on the semantic level. Abstraction eliminates the probabilistic information from a stochastic strategy by replacing the stochastic choices of actions with non-deterministic ones, and produces a strategy. Specifically, as defined in Definition 4, in the phase of verification, both strategies and stochastic ones always choose the actions with the highest scores. This is the so-called "exploitation" in reinforcement learning literature. When ties among actions appear, stochastic strategies equally likely choose one of these actions, whereas strategies make the decision non-deterministically. Therefore, a strategy may exhibit more behaviors than the stochastic strategy that the former is abstracted from. We prove this formally as follows:

Theorem 1. *Given a TG $\mathcal{G}$, an STG $\mathcal{P}$ obtained from $\mathcal{G}$ by the probabilistic quantification, a stochastic strategy σ° (Definition 4) solving $\mathcal{P}$, and a strategy σ abstracting σ°, the following inclusion holds: $Out(\mathcal{P} \mid \sigma^\circ) \subseteq Out(\mathcal{G} \mid \sigma)$.*

Proof. First, since $\mathcal{P}$ is obtained from $\mathcal{G}$ by the probabilistic quantification, an uncontrollable action that is chosen non-deterministically by $\mathcal{G}$ is chosen with equal probability by $\mathcal{P}$. If $\pi \in Out(\mathcal{P} \mid \sigma^\circ)$ and $q = last(\pi)$, there must be a $\pi' \in Out(\mathcal{P} \mid \sigma^\circ)$ such that $\pi = last(\pi') \xrightarrow{e} q$, where e meets one of the three conditions in Definition 2. Assuming $\pi' \in Out(\mathcal{G} \mid \sigma)$, then

1. if $e \in \Sigma_u$, then $last(\pi') \xrightarrow{e} q \in Out(\mathcal{G} \mid \sigma)$ because σ has no control on e, and $\mathcal{G}$ non-deterministically chooses $e \in \Sigma_u$. Hence, $\pi \in Out(\mathcal{G} \mid \sigma)$;

2. if $e \in \Sigma_c \cap \sigma(q)$ or $e = \lambda$, then according to Definition 4, e has the highest score in $\mathcal{A}^q_{\mathcal{G}}$. Then e can be chosen by σ deterministically when $\|\mathcal{A}\| = 1$ or non-deterministically when $\|\mathcal{A}\| \neq 1$. Hence, $\pi \in Out(\mathcal{G} \mid \sigma)$.

Hence, $\pi = last(\pi') \xrightarrow{e} q \in Out(\mathcal{G} \mid \sigma)$. Likewise, we can inductively prove the assumption: $\pi' \in Out(\mathcal{G} \mid \sigma)$. Hence, if $\pi \in Out(\mathcal{P} \mid \sigma^\circ)$, $\pi \in Out(\mathcal{G} \mid \sigma)$, that is, $Out(\mathcal{P} \mid \sigma^\circ) \subseteq Out(\mathcal{G} \mid \sigma)$. □

Theorem 1 shows that σ, as the abstraction of σ°, may broaden σ°'s outcome, since the former may exhibit behaviors that do not exist in the latter.

Example. Assume $\mathcal{P} \mid \sigma^\circ$ contains a state q_i having 10 controllable actions, among which 4 of them have the highest score in σ°, namely actions a_i^1, a_i^2, a_i^3, and a_i^4, respectively. Then a run as follows is possible in both $\mathcal{P} \mid \sigma^\circ$ and $\mathcal{G} \mid \sigma$,

$$... \; q_i \xrightarrow{a_i^1} q_{i+1} \rightarrow ... \rightarrow q_i \xrightarrow{a_i^2} q'_{i+1} \rightarrow ... \rightarrow q_i \xrightarrow{a_i^3} q''_{i+1} \rightarrow ... \rightarrow q_i \xrightarrow{a_i^4} q'''_{i+1} \; ...$$

where state q_i is reached 4 times when each of a_i^1, a_i^2, a_i^3, and a_i^4 gets to be chosen once. However, a run as follows is possible in $\mathcal{G} \mid \sigma$ but not in $\mathcal{P} \mid \sigma^\circ$:

$$... \; q_i \xrightarrow{a_i^2} q'_{i+1} \rightarrow ... \rightarrow q_i \xrightarrow{a_i^2} q'_{i+1} \rightarrow ... \rightarrow q_i \xrightarrow{a_i^2} q'_{i+1} \rightarrow ... \rightarrow q_i \xrightarrow{a_i^2} q'_{i+1} \; ...$$

where a_i^2 is chosen 4 times in row. Therefore, the post-verification on σ is necessary for ensuring that the resulting strategy meets the requirements. In the next section, we enumerate other reasons for conducting the post-verification.

4.4.3. Strategy Synthesis

Synthesis in UPPAAL STRATEGO is done via the following queries:

```
strategy policy = minE(x)[<=T]{dv}-->{cv}:<> P          (3)
```

```
strategy policy = maxE(x)[<=T]{dv}-->{cv}:<> P          (4)
```

The keyword `minE(x)` (respectively, `maxE(x)`) means simulating the model while running the learning algorithm with the purpose of minimizing (respectively, maximizing) "`x`", which can be a variable or an expression. This is the so-called "reward function" in reinforcement learning literature. In addition, `T` is the maximum time for one round simulation, `dv` is a set of expressions regarded as discrete values, and `cv` is a set of expressions regarded as continuous values. These constitute the so-called "features" in reinforcement learning literature [21].

The state space of the MAS TG is partially shown to the learning algorithm by the values of the expressions in `dv` and `cv`. In particular, MoCReL only allows discrete variables, hence the synthesized strategies do not contain clocks. This limitation facilitates the verification of the learned strategy since the preference of choice of controllable action cannot change within zones that represent the basic construction enabling symbolic verification of timed automata [15].

The formula "`<> P`" is a CTL property, and only the runs that satisfy this property are sampled in the simulation. These runs are used as input of the learning algorithm to calculate the scores of state-action pairs. In particular, MoCReL uses "`<> time` $\geq$ `C`", where `time` is a global clock in the model that is never reset and `C` $\in [0,$ `T`$]$ is a constant integer within the simulation time `T`. This formula allows all runs that simulate the model over `C` time units to be passed to the learning algorithm no matter whether the agents reach their goals or not. Hence, both good and bad state-action pairs are passed to the learning

algorithm, which accumulates their scores by using their rewards or penalties, respectively.

When running Query (3) in UPPAAL STRATEGO, our new version of the tool calls an external library, which implements the learning algorithm of MoCReL to synthesize strategies, and stores the score table of the strategy. With the help of the external library, one can plug in one's own learning algorithm or add new functions into the existing algorithm. We show this in Section 4.4.5.

Example. Now, we revisit the path-finding problem of Section 3.2, Fig. 2, to shown on the example concretely the necessity of verifying the resulting strategies, which in fact follows from the one way inclusion of Theorem 1. Assume that the cat stays at its current position for `N` minutes, and that the robot wants to catch it as quickly as possible, then the reward function can be specified as:

$$\texttt{x} = \texttt{time} - \texttt{caught} \times \texttt{REWARD} \tag{5}$$

The variable `time` is the global clock aforementioned, `caught` is a binary integer (i.e, 1/0) indicating if the cat is caught by the robot or not, and `REWARD` is a non-negative integer that the robot gets when it catches the cat. It is trivial to see that the smaller the value of `x` is, the better the strategy is.

If one mistakenly adopts the reward function of equation 5 but Query (4) for synthesis, which attempts to find the state-action pairs maximizing `x`, the result can still be obtained, as the synthesis is only about accumulating scores of the pairs and populating a score table. However, the actions that consume the longest time (i.e., `time` being maximum) but never catch the cat (i.e., `caught` being 0) are taken as the best actions in this result.

This example shows a possible misuse of the queries for synthesis. Even if one uses the query and reward function correctly, the resulting mission plan may still be wrong, because the samples for learning may be too few to populate a score table that covers enough states, or the MAS model is wrongly designed and violates other requirements of the agents that are not reflected in the queries for synthesis. In a nutshell, the learning-based synthesis does not have a correctness-guarantee on its results.

4.4.4. Strategy Verification

Different from MCRL [12], the verification in MoCReL is directly conducted on the MAS model under the control of a strategy, because Uppaal Stratego supports the following verification queries [14]:

$$\texttt{A<>}\ \phi\ \texttt{under}\ \sigma \tag{6}$$

$$\texttt{Pr[<=T]}\ \phi\ \texttt{under}\ \sigma \tag{7}$$

The keyword `under` puts the state space exploration of the MAS TG under the control of the strategy that is synthesized and stored by the external library of MoCReL. Query (6) returns an absolute answer of true or false to the question of whether ϕ is always eventually satisfied, whereas Query (7) returns the probability of satisfying ϕ.

In this paper, we extend Uppaal Stratego to support Query (6) on strategies that are synthesized by learning. The pseudo-code of executing Query (6) is in Algorithm 1. In Appendix A.2, we illustrate the execution of the algorithm with an example. Here, we overview the algorithm briefly. To verify a liveness property like Query (6), one needs to explore the model's state space until either getting a counter-example run violating the property, or until reaching all the states. Specifically, a counter-example of a liveness property like Query (6) must be either a loop, or a maximum run ending at an unbounded state or a deadlock, in which all the states do not satisfy ϕ. Hence, once such a run is found, the verification terminates with a negative result (line 15 and line 17).

Additionally, the state space exploration must be guided by a strategy. When the model checker faces controllable actions (i.e., in line 22, $isControllable(\stackrel{a}{\Longrightarrow})$ returns *true*)), or a delay (lines 7 and 9), it calls a function `Allow` to lookup the score table of a strategy. According to Definition 4, actions with the highest scores are always chosen – with ties broken by a uniform distribution (Query (7)) or a non-deterministic choice (Query (6)). In this way, the liveness verification is guided by a strategy.

Example. We show several queries that can be used in the verification of the

Algorithm 1: Algorithm of liveness verification (adapted from Fig. 3 in the literature [27]): model checking $\mathcal{G} \mid \sigma$ against Query (6)

```
1  Function Liveness(𝒢, σ, φ):
2      ST := ∅ SD := ∅ Passed := ∅
3      Delay(𝒢.S_0, ¬φ)
4      for S_d ∈ SD do
5          Search(S_d, ¬φ)
6      return (true)
7  Function Delay(S, φ):
8      for S' : S ⟹^d S' do
9          if Allow(σ, ⟹^d) then
10             if (S' ∉ SD) ∧ (S' ⊨ I(S.l) ∧ φ) then
11                 push(SD, S')
12 Function Search(S, φ):
13     S := S ∧ φ
14     if S ≠ empty then
15         if loop(S,ST) then
16             exit(false)                                   // Loop found
17         if unbounded(S) ∨ deadlocked(S) then
18             exit(false)                                   // Maximal run found
19         push(ST, S)
20         if ∀S' ∈ Passed : S ⊈ S' then
21             for S_a : S ⟹^a S_a do
                       // If action a is uncontrollable or allowed, it can be chosen.
22                 if ¬isControllable(⟹^a) ∨ Allow(σ, ⟹^a) then
23                     Delay(S_a, φ)
24                     for S_d ∈ SD do
25                         Search(S_d, φ)                    // Recursive all
26     Passed := Passed ∩ {pop(ST)}                          // Move from stack to Passed
27 Function Allow(S, action):
28     if NumControllable(S) == 1 then
29         return (true)
30     if action ∈ best(σ, S) then
31         label(σ, S, action)                               // Label (S, action) in σ
32         return (true)
33     else
34         return (false)
```

synthesized strategy in the path-finding problem of Section 3.2, Fig. 2.

$$\texttt{strategy policy = minE(time - caught} \times \texttt{REWARD)[<=100]} \\ \texttt{\{robot.location\}-->\{\}:<> time} \geq \texttt{90} \tag{8}$$

$$\texttt{A<> caught under policy} \tag{9}$$

$$\texttt{A[] !collide() under policy} \tag{10}$$

Query (8) synthesizes a strategy named `policy`, which is supposed to catch

the cat within 100 time units. Query (9) verifies the robot model under the control of `policy` to see if it can always eventually catch the cat. Query (10) involves a function `collide()` implemented in the model, which detects the distances from the robot to obstacles in the environment and returns *true* if any one of the distances is less than a certain value, or *false* otherwise. This query verifies that the collision between the robot and obstacles never happens.

Besides the possible errors in the resulting strategies, as presented in the path-finding example, strategies can be memory consuming for containing too many useless data. With the help of the external library where MoCReL is implemented, we can leverage queries in the form of Query (6) to not only verify the strategy but also compress the strategy.

4.4.5. Strategy Compression

Once an external library is linked to Uppaal Stratego, the model checker always enquires the external library when facing multiple controllable actions. For example, when more than one agent is ready to execute a task, the model checker without an external library simply traverses all options non-deterministically, whereas the model checker with an external library passes the current state and the available actions of the state to the external library one by one, and obtains the preference of each state-action pair. The ones with the highest score are always preferred. In MoCReL, besides returning the preference, we also label the state-action pairs that have the highest score as "selected" because they will be selected and verified by the model checker.

When verifying a liveness property (e.g., Query (6)), the model checker must explore all the branches of the state space to ensure that the proposition of the property (e.g., ϕ in Query (6)) is always eventually *true*. Therefore, if the liveness property is satisfied, the labelled state-action pairs are "selected" from the state space and the exhaustiveness of search guarantees them to always eventually reach the states where the property is true. The unlabelled pairs are considered "useless" data because without them, the property can still be satisfied. Therefore, the strategy can be compressed by removing the unlabelled

Algorithm 2: MoCReL algorithm

```
1  Function Main(G, Q, iterationNum, totalNum, goodNum, formula):
2      Strategy σ := ∅, σ_c := ∅
3      Stochastic Strategy σ° := ∅
4      STG P := ProbabilisticQuantification(G)
5      while ¬Liveness(G, σ, Q) do
6          σ° := Learn(P, iterationNum, totalNum, goodNum, formula)
7          σ := Abstraction(σ°)
8          Update(iterationNum, totalNum, goodNum)
9      σ_c := Clean(σ)
10     return (σ_c)
```

pairs (*cleaning* in Fig. 4). By verifying the compressed strategy again, we can see that the new strategy preserves the liveness property that is met by the original strategy.

4.4.6. Soundness of MoCReL

Algorithm 2 is the pseudo-code of MoCReL. Line 4 and line 7 are the probabilistic quantification and abstraction, respectively. Line 6 runs an algorithm that iteratively simulates and learns until a user-defined number of samples are obtained, or the iteration reaches its maximum rounds (see Algorithm 3 in Appendix A.1). The function `Liveness`$(\mathcal{G}, \sigma, \mathcal{Q})$ at line 5 runs Algorithm 1, which verifies if $\mathcal{G} \mid \sigma \models \mathcal{Q}$ as defined in Definition 3, and labels the state-action pairs that are selected by the model checker. Line 8 updates the parameters for learning, e.g., increasing the number of samples (i.e., `totalNum`) to have a larger score table that covers more states than that of the last strategy. Line 9 compresses σ by removing the unlabeled data.

Soundness of the Approach. When MoCReL terminates with a synthesized strategy, the result is verified, which guarantees that the planning problem (Definition 3) is answered correctly. Formally, MoCReL is sound, proven by Theorem 2 below:

Theorem 2 (Soundness)**.** *Given a planning problem $Q =< \mathcal{G},\ \mathcal{Q} >$, where $\mathcal{Q} = A <> \phi$, if Algorithm 2 terminates and returns a strategy σ_c, then $\mathcal{G} \mid \sigma_c \models \mathcal{Q}$.*

Proof. Obviously, Algorithm 2 terminates with two cases:

1. `Liveness` returns *true* (line 5 in Algorithm 2), when Algorithm 2 will eventually return σ_c (line 10 in Algorithm 2);

2. `Liveness` exits with a negative result (line 16 and line 18 in Algorithm 1), and no strategy is returned (line 10 in Algorithm 2 never being reached).

In Case 2, no strategy is generated, hence, we only need to prove when Case 1 happens, $\mathcal{G} \mid \sigma_c \models A <> \phi$. Assuming `Liveness` returns *true*, but $\mathcal{G} \mid \sigma_c \not\models A <> \phi$, then $\mathcal{G} \mid \sigma_c \models E[]\neg\phi$, which holds if and only if the following two conditions hold (the code lines referred to in the rest of the proof are all of Algorithm 1):

(i) The labeling is complete, that is, all the controllable state-action pairs that are selected by the model checker are labeled, but $\mathcal{G} \mid \sigma_c \models E[]\neg\phi$, which reads that there exists a run in $\mathcal{G} \mid \sigma_c$, in which all the states do not satisfy ϕ;

(ii) The labeling is incomplete, that is, some pairs that are selected by the model checker are not labeled, which makes the model checker use the wrong actions at certain states when verifying $\mathcal{G} \mid \sigma_c \models A <> \phi$ and get a negative result.

In Case (i), such a run is either a loop or a run ending in a deadlock or an unbounded state, in which all the states do not satisfy ϕ. Then the `Search` function must exit with a verification result of false (line 16 and line 18), which contradicts that `Liveness` returns true assumption.

In Case (ii), wherever the model checker faces a controllable action (line 22) or a delay (lines 7 and 9), it invokes the function `Allow`, which returns true when the state has only one controllable action (line 28), or the action is labeled (line 31). Hence, when facing multiple controllable actions, the model checker can never select an unlabeled action. Therefore, Case (ii) cannot happen.

In a nutshell, Case (i) and Case (ii) cannot happen, and thus $\mathcal{G} \mid \sigma_c \models E[]\neg\phi$ does not hold, that is, $\mathcal{G} \mid \sigma_c \models A <> \phi$ must hold when the function `Liveness` returns true, that is, when Algorithm 2 terminates and returns σ_c. □

5. Experimental Evaluation

In this section, we evaluate MoCReL in several experiments to see its performance in the use case of an autonomous quarry with different numbers of agents, tasks, and task execution time. The reinforcement learning algorithm used in the experiments is Q-learning [22]. The experiments are conducted on an Intel Xeon E5-2678 with 256 GB of RAM running Ubuntu 20.04 LTS. All the models, tool, and the full experiment results can be found at: https://github.com/rgu01/MoCReL-Experiments.git.

5.1. Use Case Description

Fig. 10 depicts an autonomous quarry that is abstracted from a real scenario, where there are two kinds of autonomous agents: wheel loaders and trucks. Wheel loaders dig stones and load them into trucks. The latter load stones either from the wheel loaders or from a primary crusher, before transporting the stones to their destination: a secondary crusher. The goal of the agents is to transport a certain amount of stones. Agents need to go to a charging station for refueling when the energy level is low.

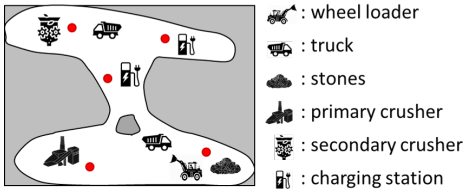


Figure 10: An autonomous quarry

To solve the MAS planning problem in this use case, first we model the system in the way described in Section 4.2. For simplicity, the subproblem of path finding is solved by the A* algorithm [3] and our movement TG only models the traveling between every pair of milestones where tasks are carried out (e.g., red dots in Fig. 10). Although simplified, the state space of the problem during verification still grows exponentially with the linear increase of the number of agents [9]. Task execution TG models four types of tasks: (i) individual tasks with no precondition, e.g., wheel loaders digging stones; (ii) individual tasks with preconditions, e.g., trucks unloading stones into the secondary crushers with a precondition: the unloading task can be carried out only

after the trucks have been loaded by wheel loaders or at primary crushers; (iii) collaborating tasks, e.g., wheel loaders loading stones into trucks; (iv) tasks that are activated by events, e.g, refueling when an agent's energy level is low. In addition, we design a special TG named *Referee* (Fig. A.16 in Appendix A.3), which judges if the goal is reached (i.e., enough stones are transported) or the maximum simulation time has been reached. In either case, the agents must stop, i.e., no controllable actions can be taken. The learning algorithm partially observes the state space of the models by detecting discrete variables such as the locations of the TG[4].

According to our previous study, the method that purely uses search-based algorithms can only solve a simplified version of this problem, where task execution time is fixed and the number of agents is less than 5 [9, 13]. MCRL [12] can deal with more agents and flexible task execution time, but collaborations and events are not supported. These experiments include the collaboration among agents and a battery-low event. Maps in the experiments are also complex, i.e., some models contain 2-4 primary crushers and 1-2 secondary crushers.

5.2. Experiment Design

We conduct two series of experiments: 1) one where we study the synthesis time and compression efficiency, and 2) one where we study the influence of the number of sampled runs on the learning efficiency. Models that are used in both series of experiments are generated automatically by randomly assigning values to the parameters of the environment, e.g., the number of agents. The parameters are reported in Table 1 that we introduce in the next sub-section. The abbreviations in Table 1 are given as a footnote[5].

The first series of experiments is conducted on the full set of models while

[4]Discrete variables can be seen in the queries of the models in the artifacts.

[5]Abbreviations in Table 1: category (CAT), the number of wheel loaders (WL), the number of trucks (TK), the number of primary crushers (PC), the number of secondary crushers (SC), the number of chargers (CH), the capability of trucks (CAP), if the task execution time is time intervals or not (INT), the number of runs (RUNS), the computation time of synthesis in seconds (STIME), the size of the original strategy in MB (ORI), the size of the compressed strategy in MB (COM), the result of verification (VER).

Table 1: Results of strategy synthesis, verification, and compression

CAT	model	WL	TK	PC	SC	CH	CAP	INT	RUNS	STIME	ORI	COM	VER
I	game1-A	2	4	1	1	0	20	YES	2000	3,902	27	0.13	TRUE
	game3-A	1	2	1	1	0	20	YES	200	16	0.08	0.02	TRUE
	game4-A	2	4	1	1	0	20	YES	5,000	772	5.6	0.03	TRUE
	game6-A	2	1	1	1	0	20	YES	200	175	0.09	0.02	TRUE
	game7-A	1	4	1	1	0	20	YES	5,000	575	4.7	0.03	TRUE
	game8-A	1	2	1	1	0	20	YES	200	14	0.08	0.02	TRUE
	game9-A	1	4	1	1	0	20	YES	5,000	640	4.4	0.05	TRUE
II	game0-B	1	2	3	1	0	10	YES	500	92	0.9	0.2	TRUE
	game1-B	1	1	4	1	0	10	YES	500	71	0.02	0.1	TRUE
	game3-B	1	2	1	2	0	10	YES	100,000	17,297	1.4	0.6	TRUE
	game1-E	1	3	1	2	0	30	NO	500	88	5.9	0.03	TRUE
	game5-E	1	3	4	2	0	30	NO	5000	1,705	103	0.05	TRUE
	game2-B	1	4	1	2	0	10	YES	100,000	800	112	-	FALSE
	game6-B	1	3	3	2	0	10	YES	100,000	893	121	-	FALSE
III	game4-C	1	2	1	1	2	50	YES	2,000	270	9.4	0.03	TRUE
	game5-C	1	2	1	1	1	50	YES	5000	410	2.8	0.03	TRUE
	game3-D	1	2	1	1	1	50	NO	500	68	1.4	0.03	TRUE
	game6-D	1	2	1	1	2	50	NO	500	80	2.6	0.03	TRUE
	game9-D	1	2	1	1	2	50	NO	500	84	7.0	0.03	TRUE
	game6-C	1	1	1	1	2	50	YES	100,000	8,629	0.7	-	FALSE
	game8-C	1	2	1	1	2	50	YES	100,000	12,457	49	-	FALSE

the second is restricted to a subset. The set of models is grouped intro three categories:

- *Category I*: Models with large numbers of agents up to 6, a small number of crushers (2), a fixed medium value of the trucks' capabilities (20), and no *monitor* TG for charging.
- *Category II*: Models with medium numbers of agents (2 - 5), large numbers of crushers (3 - 6), a range of the trucks' capabilities (10 - 30), and no *monitor* TG for charging.
- *Category III*: Models with small numbers of agents (2 - 3) and crushers (2), a fixed large value of the trucks' capabilities (50), and 1 - 2 *monitors* TG for charging.

The second series of experiments is conducted on a model `game6-B` in Table1 and its variants that change the amount of stones trucks can carry at one time. For these three models, we modify the "RUNS" from 100 to 500, and for each number of "RUNS", we synthesize a strategy and statistically verify its probability of reaching the goal by using queries in the form of Query (7). We repeat this

experiment 10 times and use the mean values of the probabilities to be the result of verification to account for the random nature of statistical model checking.

5.3. Experiment Results

In Table 1, column "CAP" indicates the amount of stones that trucks can transport at one time, and the target amount of stones to be transported is the same in all models. Column "VER" shows the results of verifying queries in the form of Query (6). Column "RUNS" includes the numbers of runs that are needed to synthesize a strategy, which are picked empirically.

Synthesis time. In category I, the time of synthesizing strategies is relatively short, respectively. Most of the cases spend several seconds and the most difficult one (`game1-A`) costs more than 1 hour with the largest strategy (27M) produced in this category. In category II, synthesis time remains at the level of minutes for most of the cases. One interesting comparison is between `game3-B` and `game5-E` in this category. Considering the numbers of agents and milestones (e.g., crushers), the latter is more complex than the former. However, `game3-B` needs 100,000 runs and more than 4 hours to synthesize a successful strategy that passes the verification, whereas `game5-E` only needs 5000 runs and half an hour. The reason is because the task execution times are fixed in `game5-E` whereas the ones in `game3-B` are time intervals. The time intervals cause many interleaving actions which increase the state space of the model dramatically. When maps have chargers in category III, the synthesis times for successful strategies are at most several minutes. However, some models in categories II and III can be very complex so that learning with $100,000$ runs cannot generate successful strategies. We will discuss this in the presentation of learning efficiency.

Verification results. Overall, most of the cases ($\frac{41}{50}$) in the experiments pass the verification[6]. In some cases (e.g. `game2-B` in category II), we find counter-examples in the strategies that violate the liveness property, so they do

[6]Full results of all models can be seen: shorturl.at/dkqyE

not pass the verification. Increasing their simulation time and rounds to gather more runs for learning can be helpful in these cases. However, the fact that the models in these cases have large state spaces makes reaching the goal state a rare event that is hard to catch by random simulation (see the results of learning efficiency). This phenomenon stems from the nature of reinforcement learning algorithms that rely on random simulation.

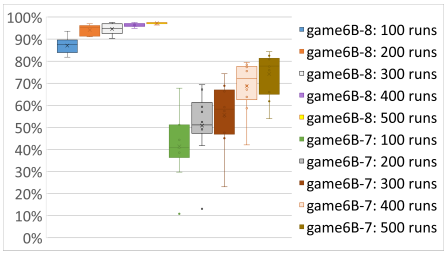


Figure 11: Distribution of mean probabilities of satisfaction over 10 experiments

Learning efficiency. Fig. 11 shows the mean probabilities of agents reaching their goal (i.e., satisfying Query (7)). The original model is `game6B`, in which the capability of trucks is 10, and the modified models are `game6B-7` and `game6B-8`, which decrease the capability to 7 and 8, respectively. The results of model `game6B` are not shown in the figure because all the experiments with 100 to 500 runs generate the same result: above 97%. The probabilities of `game6B-7` and `game6B-8` increase with the increasing numbers of runs. The probabilities of model `game6B-7` are lower than those of the other two models and the IQR (interquartile ranges[7]) are the largest. This indicates that when reaching the goal becomes hard, learning efficiency becomes unstable in the sense that the probabilities of satisfaction under the learned strategy vary dramatically.

One interesting observation is that, although the original model of `game6B` cannot generate a successful strategy not even when the number of runs is 100,000, its mean probabilities of satisfaction for the strategies synthesized by a few runs (i.e., 100 - 500) are quite high (i.e., above 97%) with a standard deviation of 0. This phenomenon shows that when reaching the goal becomes a rare event, the benefit of increasing the number of runs becomes very low.

Strategy compression. The reduced sizes of compressed strategies are up

[7]IQR is the difference between the 75th and 25th percentiles of the data.

to 99.95% of the original sizes in our experiments (e.g., `game5-E` in category II). Strategies that do not pass the verification are not compressed and thus are shown as "-" in the column "COM" of Table 1. The compressed strategies not only save memory space but also improve the explainability of the strategies. For example, the score table of the complete strategy in `game4-A` has almost 78,000 rows of data, which is reduced to less than 50 rows in the compressed strategy[8]. The latter is much more readable and explainable by humans.

Conclusion of the Experiments. The experiments show that MoCReL can solve the MAS planning problem in complex maps with multiple crushers and chargers. Successful strategies are verified and compressed and the reduced sizes are significant. Counter-examples of the liveness property can be found in unsuccessful strategies, which indicate where the agents fail. Compared to MCRL, although the environment is more complex, the task types are richer, and the number of agents is larger, MoCReL can still solve most of the cases in a reasonable time. The learning efficiency of reinforcement learning drops dramatically when reaching the goal state becomes a rare event in the model.

6. Related Work

Synthesis of strategies for MAS has been an increasingly researched area in recent years. Andersen et al. [28] present a UPPAAL-based method for motion planning of multi-robot systems. Their method uses reachability queries to generate motion plans, which is not sufficient for synthesizing comprehensive strategies that consider time intervals as the execution time of motions. Gleirscher et al. [29] introduce an approach for synthesis and verification of safety controllers for human-robot collaboration. The main difference between our work and theirs is that their synthesis is based on search, which is correct-by-construction, but the scalability is limited. Bouton et al. [30] propose a method that enforces probabilistic guarantees on agents during the course of reinforce-

[8]Please see the printed strategies of `game4-A` in the artifacts.

ment learning, whereas our method provides post-verification and compression of the synthesized strategies.

In the area of strategy compression, Julian et al. explore several ways of compressing strategies by using origami compression [31] or deep neural network [23][32]. Ashok et al. propose a decision-tree-based method for concisely representing strategies [33][34]. Their tool *dtControl* is able to compress strategies produced by UPPAAL TIGA. Compared with these methods, the strategy compression in MoCReL focuses on cleaning the unused data in the strategies rather than representing them in different forms. Compression in MoCReL replying on exhaustive model checking inherently provides safety guarantee of the strategies, which needs extra effort to achieve in other methods [32].

7. Conclusions and Future Work

We present a new method, namely MoCReL, for synthesis, verification, and compacting of strategies of multi-agent autonomous systems (MAS). MoCReL uses reinforcement learning for synthesizing strategies and model checking for verifying and compressing the strategies. MoCReL is integrated into UPPAAL STRATEGO, which facilitates the use of this method. Experiments carried out on a real-word autonomous quarry case study show that MoCReL is able to solve the planning problem of MAS in complex maps with large numbers of agents. The compressed strategies save up to 99.95% of the memory space taken by the original strategies. When reaching the goal state becomes a rare event that is hard to be captured by random simulation, the learning efficiency of reinforcement learning drops dramatically.

An interesting direction of the future work is to investigate the use of the counter-examples to repair the unsuccessful strategies, which would increase the learning efficiency profoundly. Introducing clocks into the strategies can be another challenging direction of research.

Acknowledgments

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

References

[1] E. Oliveira, K. Fischer, O. Stepankova, Multi-agent systems: which research for which applications, Robotics and Autonomous Systems 27 (1-2) (1999) 91–106.

[2] P. Chandler, M. Pachter, Research issues in autonomous control of tactical uavs, in: Proceedings of the 1998 American Control Conference. ACC (IEEE Cat. No. 98CH36207), IEEE, 1998.

[3] S. Rabin, Game programming gems, chapter a* aesthetic optimizations, Charles River Media (2000).

[4] S. M. LaValle, Rapidly-exploring random trees: A new tool for path planning, in: Technical Report, 1998.

[5] J.-F. Kempf, M. Bozga, O. Maler, 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, pp. 385–400.

[6] K. G. Larsen, A. Le Coënt, M. Mikučionis, J. H. Taankvist, Guaranteed control synthesis for continuous systems in uppaal tiga, in: Cyber Physical Systems. Model-Based Design, Springer, 2018, pp. 113–133.

[7] W. Zhang, T. G. Dietterich, High-performance job-shop scheduling with a timedelay td () network, Advances in neural information processing systems 8 (1996) 1024–1030.

[8] C. Shyalika, T. Silva, A. Karunananda, Reinforcement learning in dynamic task scheduling: A review, SN Computer Science 1 (6) (2020) 1–17.

[9] R. Gu, E. P. Enoiu, C. Seceleanu, Tamaa: Uppaal-based mission planning for autonomous agents, in: 35th ACM/SIGAPP Symposium On Applied Computing SAC2020, ACM, 2019.

[10] M. Bouton, A. Cosgun, M. J. Kochenderfer, Belief state planning for autonomously navigating urban intersections, in: Intelligent Vehicles Symposium, IEEE, 2017.

[11] Y. Abdeddaı, E. Asarin, O. Maler, Scheduling with timed automata, Theoretical Computer Science 354 (2) (2006) 272–300.

[12] R. Gu, E. P. Enoiu, C. Seceleanu, K. Lundqvist, Verifiable and scalable mission-plan synthesis for multiple autonomous agents, in: 25th International Conference on Formal Methods for Industrial Critical Systems, Springer, 2020.

[13] R. Gu, P. Jensen, D. Poulsen, C. Seceleanu, E. Enoiu, K. Lundqvist, Verifiable strategy synthesis for multiple autonomous agents: A scalable approach, International Journal on Software Tools for Technology Transfer (STTT) 24 (3) (2022).

[14] A. David, P. G. Jensen, K. G. Larsen, M. Mikučionis, J. H. Taankvist, Uppaal stratego, in: TACAS 2015: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2015.

[15] J. Bengtsson, W. Yi, Timed automata: Semantics, algorithms and tools, Lectures on Concurrency and Petri Nets: Advances in Petri Nets (2004) 87–124.

[16] A. David, D. Du, K. G. Larsen, A. Legay, M. Mikučionis, D. B. Poulsen, S. Sedwards, Statistical model checking for stochastic hybrid systems, arXiv preprint arXiv:1208.3856 (2012).

[17] F. Cassez, A. David, E. Fleury, K. G. Larsen, D. Lime, Efficient on-the-fly algorithms for the analysis of timed games, in: CONCUR 2005: International Conference on Concurrency Theory, Springer, 2005, pp. 66–80.

[18] R. Alur, D. L. Dill, A Theory of Timed Automata, Theoretical Computer Science 126 (1994) 183–235.

[19] A. David, P. G. Jensen, K. G. Larsen, A. Legay, D. Lime, M. G. Sørensen, J. H. Taankvist, On time with minimal expected cost!, in: International Symposium on Automated Technology for Verification and Analysis, Springer, 2014, pp. 129–145.

[20] C. Baier, J.-P. Katoen, Principles of model checking, MIT press, 2008.

[21] R. S. Sutton, A. G. Barto, Reinforcement learning: An introduction, MIT press, 2018.

[22] C. J. C. H. Watkins, Learning from delayed rewards, King's College, Cambridge United Kingdom, 1989.

[23] K. D. Julian, M. J. Kochenderfer, M. P. Owen, Deep neural network compression for aircraft collision avoidance systems, Journal of Guidance, Control, and Dynamics 42 (3) (2019) 598–608.

[24] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to algorithms, MIT press, 2009.

[25] M. Jaeger, P. G. Jensen, K. G. Larsen, A. Legay, S. Sedwards, J. H. Taankvist, Teaching stratego to play ball: Optimal synthesis for continuous space mdps, in: International Symposium on Automated Technology for Verification and Analysis, Springer, 2019, pp. 81–97.

[26] G. Behrmann, A. David, E. Fleury, K. Larsen, D. Lime, E. Nantes, Uppaal-Tiga: Time for playing games! (tool paper), in: Proceedings of the 2007 Computer Aided Verification, Springer Berlin Heidelberg, 2007.

[27] G. Behrmann, K. G. Larsen, J. I. Rasmussen, Beyond liveness: Efficient parameter synthesis for time bounded liveness, in: International Conference on Formal Modeling and Analysis of Timed Systems, Springer, 2005, pp. 81–94.

[28] M. S. Andersen, R. S. Jensen, T. Bak, M. M. Quottrup, Motion planning in multi-robot systems using timed automata, IFAC Proceedings Volumes 37 (8) (2004) 597–602.

[29] M. Gleirscher, R. Calinescu, J. Douthwaite, B. Lesage, C. Paterson, J. Aitken, R. Alexander, J. Law, Verified synthesis of optimal safety controllers for human-robot collaboration, arXiv preprint arXiv:2106.06604 (2021).

[30] M. Bouton, J. Karlsson, A. Nakhaei, K. Fujimura, M. J. Kochenderfer, J. Tumova, Reinforcement learning with probabilistic guarantees for autonomous driving, arXiv preprint arXiv:1904.07189, 2019.

[31] K. D. Julian, J. Lopez, J. S. Brush, M. P. Owen, M. J. Kochenderfer, Policy compression for aircraft collision avoidance systems, in: 2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC), IEEE, 2016, pp. 1–10.

[32] K. D. Julian, M. J. Kochenderfer, Guaranteeing safety for neural network-based aircraft collision avoidance systems, in: 2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC), IEEE, 2019, pp. 1–10.

[33] P. Ashok, M. Jackermeier, P. Jagtap, J. Křetínskỳ, M. Weininger, M. Zamani, dtcontrol: Decision tree learning algorithms for controller representation, in: Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control, 2020, pp. 1–7.

[34] P. Ashok, M. Jackermeier, J. Křetínskỳ, C. Weinhuber, M. Weininger, M. Yadav, dtcontrol 2.0: Explainable strategy representation via decision tree learning steered by experts, arXiv preprint arXiv:2101.07202 (2021).

Appendix A. Appendix

Appendix A.1. Algorithm of Synthesis

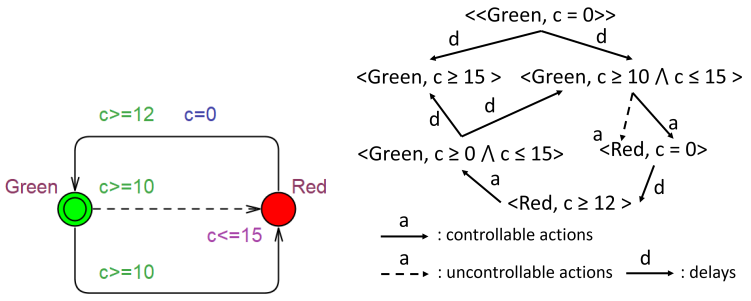
Algorithm 3 is the simplified pseudo-code of running Query (3) in UPPAAL STRATEGO. Details of this algorithm are in the literature [13].

Algorithm 3: Simplified algorithm behind the `minE`-query (adapted from Algorithm 1 in the literature [13])

```
1  minE(tg, 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(tg, 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;
```

Appendix A.2. Algorithm of Verification and Labeling

In Algorithm 1, line 3 passes the initial state S_0 of the TG $\mathcal{G}$ and the negation of the state formula of Query (6), i.e., $\neg\phi$, to the function `Delay`, which adds the symbolic succeeding states of S_0 via restricted delay transitions. The definition of restricted delay transitions is presented in literature [27]. In this paper, we adapt this function on symbolic states (i.e., zones) by using difference bounded matrices (DBM) in UPPAAL. Fig. A.12 shows an example of a UTA modeling



(a) An example of TG (b) The Zone Graph of the TG

Figure A.12: An example of a TG and its semantic model.

a traffic light and its symbolic semantic model - a Zone Graph. The action transitions and delay transitions are arrows labeled with `a` and `d`, respectively. An example of symbolic states that are used in the `Delay` function is <`Red, c=0`> in Fig. 12(b). The function `Allow`$(\sigma, \stackrel{d}{\Longrightarrow})$ checks if the delay transition is allowed by the strategy σ by calling back the external library (see Appendix A.4). Briefly, if the action is the only controllable action at state S, the function `Allow` returns *true* directly, which is the case at the initial state in Fig. 12(b); otherwise, it looks up the strategy and finds the set of the best actions that have the highest score at the current state (i.e., $best(\sigma, S)$). If the current action belongs to the set, it is allowed and we call the *label* function to label the state-action pair as *visited* (line 31).

When the delay transition is allowed in the function `Delay`, we continue to check if the succeeding state S' is not in the stack SD and satisfies the invariant at the location of the current state ($I(S.l)$) and the restriction (φ) (line 10). The restriction φ is actually $\neg\phi$, which means the state space exploration only visits the states where the state formula ϕ of Query (6) is *false*, because the verification of a liveness property aims to find a run where ϕ is *false* at all states as the counter-example. If S' satisfies the condition (line 10), it is pushed into the stack SD for further exploration. In Fig. 12(b), after delaying at the initial location, two symbolic states can be reached, which are passed to function `Search` as the value of parameter S. The restriction $\neg\phi$ is also passed to function `Search` as

the value of parameter φ.

In function `Search`, we first check if the current state S satisfies φ (line 13, which returns an empty state when φ is *false* at S, and S itself when φ is *true*). At line 15, the function checks if there is a loop in the state space by checking if the current state S is in the stack ST. If a loop exists, an unsatisfactory run (the runs where no state satisfies ϕ) is found and thus the algorithm exists with a negative result of verification; otherwise, we check if the maximum run is found (line 17). According to the definition in the literature [27], a run is maximal if either it ends in a state with no outgoing transitions, ends in a state from which an unbounded delay is possible, or is infinite. When such runs are found, no further symbolic state exists and thus the algorithm exists with a negative result of verification; otherwise, the algorithm pushes S into ST and continues to explore the unvisited states (line 20). For example, in Fig. 12(b), both succeeding states of the initial state are pushed into SD and explored by function `Search`. The state `<Green, c≥15>` ends at a deadlock, whereas the state `<Green, c≥10 ∧ c≤15>` has two actions, that is, a controllable action and an uncontrollable one. Both actions end to the same state `<Reg, c=0>`.

Similar to the function `Delay`, line 22 explores the succeeding states via controllable actions that are allowed by the strategy σ, or uncontrollable actions. If a controllable action is allowed, its succeeding states are recursively explored at line 25. For example, at the state `<Green, c≥10 ∧ c≤15>` in Fig. 12(b), we can either choose the uncontrollable action without asking the strategy, or choose the controllable action after asking the strategy, and then continue to explore the state space in the same manner.

Assume we instantiate a model of the TG in Fig. 12(a), namely `trafficLight`, and we want to verify a liveness property: `A<> trafficLight.Red`, by following algorithm 1, we will get a negative result of verification with a counter-example returned, that is, a trace from the initial state `<<Green, c=0>` to the state `<Green, c≥15>`.

Appendix A.3. Templates of the TG models

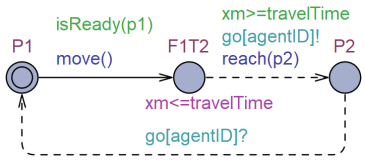


Figure A.13: The TG template of agent movement

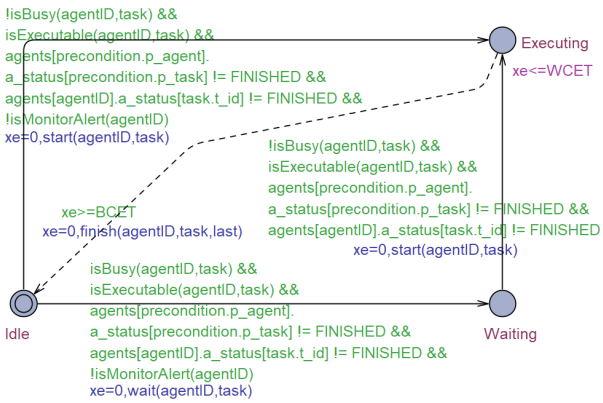


Figure A.14: A TG template of agent task execution

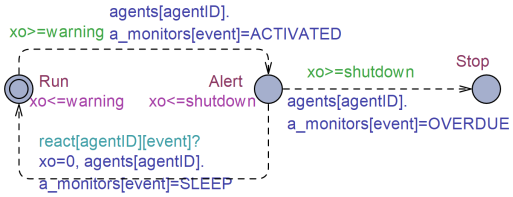


Figure A.15: The TG template of agent monitors

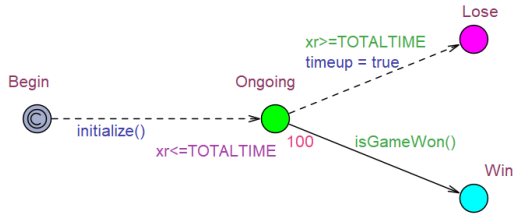


Figure A.16: The Referee TG

Appendix A.4. Overview of the External Library of MoCReL

The new extension of UPPAAL STRATEGO supports calling external libraries that are implemented by C/C++. An example of the implementation is in: `https://github.com/DEIS-Tools/stratego$_$external$_$learning`. The library must contain the following functions so that UPPAAL STRATEGO can invoke it correctly:

```
// Allocates an instance of a learner
void* uppaal_external_learner_alloc(bool minimization,
    size_t d_size, size_t c_size, size_t a_size);
// Deallocation code for object
void uppaal_external_learner_dealloc(void* object);
// print out strategies
char* uppaal_external_learner_print(void* object);
// Deep-copy function of an instance of a leaner
void* uppaal_external_learner_clone(void* object);
// Called for each sample in a trace
void uppaal_external_learner_sample_handler(void* object,
    size_t action, double* from_d_vars, double* from_c_vars,
    double* t_d_vars, double* t_c_vars, double value);
// Return the values of state-action pairs in the strategy
double uppaal_external_learner_predict(void* object, bool
    is_search, size_t action, double* d_vars, double* c_vars)
    ;
// Batch-completion call-back
void uppaal_external_learner_flush(void* object);
```

When running MoCReL in UPPAAL STRATEGO, the function `alloc` is firstly called, which instantiates the learner. Next, when Query (3) is executed, UPPAAL STRATEGO simulates the model to sample runs, which are passed to the learner by calling the function `sample_handler`. During the simulation and verification, wherever the model has more than one controllable actions, function `predict` is called for looking up the strategy and returning the value of the action at the current state. This value can be used as the probability or the

weight of choosing that action, which is introduced in Subsection 4.4. Additionally, when under verification (Query (6) is being executed), MoCReL marks the chosen state-action pairs in the function `predict` so that the strategies can be compressed after the verification passes. One can print the strategy by using a query `saveStrategy(path)` in UPPAAL STRATEGO. It will call the function `print` to print the strategy to the specific file in a standard format.