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

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 state-space explosion when the number of agents is large. Learning-based methods can alleviate this problem, but lack a guarantee of 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, Reinforcement Learning, Strategy Compression

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)

September, 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 [11, 13, 14]. 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). MoCReL provides functions of synthesizing, verifying, and compressing plans, and it relies on modeling MAS as *(Stochastic) Timed Games* in Uppaal Stratego [15], which is a tool that incorporates a symbolic model checker Uppaal [16], a statistical model checker Uppaal SMC [17], a solver for *Timed Games* Uppaal TiGa [18], and solvers for *Stochastic Timed Games* relying on learning algorithms [15]. 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 a user-defined number of execution traces of the model, which record the executed action at each state of the model and the corresponding reward. Then the learning algorithm uses these traces to synthesize a plan, which is used in the next round of simulation. This iteration of simulation and learning ends when reaching a user-defined maximum round of iteration rounds, or a user-defined number of traces are sampled so that a final plan is considered to be generated. 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 addition, to overcome the difficulty of building the models manually, we propose parameterized templates of models. UPPAAL [16] provides a rich language for defining templates in the form of extended *Timed Automata*, consisting of locations and edges, and possibly local declarations and parameters. A template is then instantiated by a process assignment. UPPAAL STRATEGO [15] inherits UPPAAL's template-based way of modeling and extends the templates to support *Timed Games*. Thanks to this feature of UPPAAL STRATEGO, our agent models are instantiated from several templates of Timed Games. Our design of the templates enables users to easily adapt the models without re-constructing the templates according to their own applications where the environment or the goal of the mission changes. For example, if the number of agents or a precondition of a task is changed, one only needs to pass a different value of the corresponding parameter of the templates. In our experiments (Section 5), we leverage this contribution to build a tool for automatic model generation.

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

(i) Parameterized model templates that are easy to adapt according to different applications.

(ii) The model templates allow for various 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 are correct-by-construction.

(iv) Experiments of MoCReL on a real industrial case study show 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 discuss the assumptions made by MoCReL and limitations of the approach that show the potential future work. In Section 7, we compare to related work, and conclude the paper in Section 8, 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 [19]. The variables model the logic clocks in systems, which are zero initially and then increase synchronously with the same rate. Uppaal [16] is a tool for modeling, simulation, and model checking of Uppaal *timed automata* (UTA), which is an extension of TA with data variables, etc. A UTA is defined as a tuple:

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

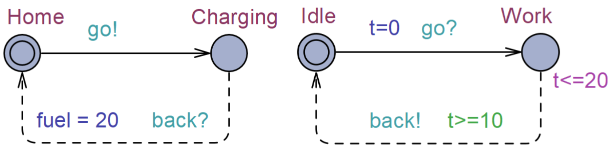
where L is a finite set of *locations*, $l_0 \in L$ is the *initial location*, Σ is a set of *actions*, V is a set of *data* variables, C is a set of real-valued variables called *clocks*, $E \subseteq L \times B(C,V) \times \Sigma \times 2^C \times L$ is the set of *edges*, where $B(C,V)$ is the set 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 $I : L \mapsto B(C)$ 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 $c' \models I(l')$.

2.2. Timed Games



(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 UTA with its set of actions partitioned into *controllable* (Σ_c) and *uncontrollable* (Σ_u) ones. Uppaal Stratego [15] 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) [16]. 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 (delays with no specific duration are denoted by λ), to each of the states. Formally, strategies are defined as follows [20]:

Definition 1 (Strategy). *Let $\mathcal{G} =< L, l_0, \Sigma_c \cup \Sigma_u, V, C, E, I >$ 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 [20]. Strategies defined by Definition 1 can be *memoryless*, which make decisions on actions depending on the current state only, that is, the function σ now is: $last(\pi_f) \rightarrow 2^{\Sigma_c \cup \{\lambda\}} \setminus \{\emptyset\}$. Note that strategies in the context of Timed Games conventionally are defined with history to capture more complex settings (such as partial observability) or complex logical requirements (such as Linear Temporal Logic). However, for the given subclass of problems that we study here (reachability and safety), memoryless strategies suffice (both w.r.t. qualitative safety/reachability and quantitative measures such as optimality), and thus it is sufficient to consider the last state.

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 $\epsilon \in Out(\mathcal{G} \mid \sigma)$ with $last(\epsilon) = q_0 = (l_0, c_0) \in Out(\mathcal{G} \mid \sigma)$[2], if $\pi \in Out(\mathcal{G} \mid \sigma)$ and $\pi' = last(\pi) \xrightarrow{e} q$, then $\pi' \in Out(\mathcal{G} \mid \sigma)$ 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 $\mathcal{G}$, then 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. In this paper, we aim to synthesize *winning*, *memoryless*, and *non-lazy* strategies, that is, winning strategies that urgently decide on a controllable action to execute, or *wait* until the environment makes a move[3]. For brevity, strategies referred to in the rest of this paper are all *memoryless* and *non-lazy.*

[1] Definition 2 is adapted from the definition of strategy outcome $Out(\sigma)$ [20].

[2] An empty trace denoted by ϵ is a special case of a trace.

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

2.3. Model Checking and Temporal Properties

Model checking [21] traverses the state space of a formal model (e.g., TA) and checks if it satisfies temporal properties. The properties in Uppaal are (Timed) Computation Tree Logic ((T)CTL) expressions [22]. In this paper, we use the following forms of (T)CTL properties, where p is an atomic proposition over the locations, clocks, and data variables of the UTA:

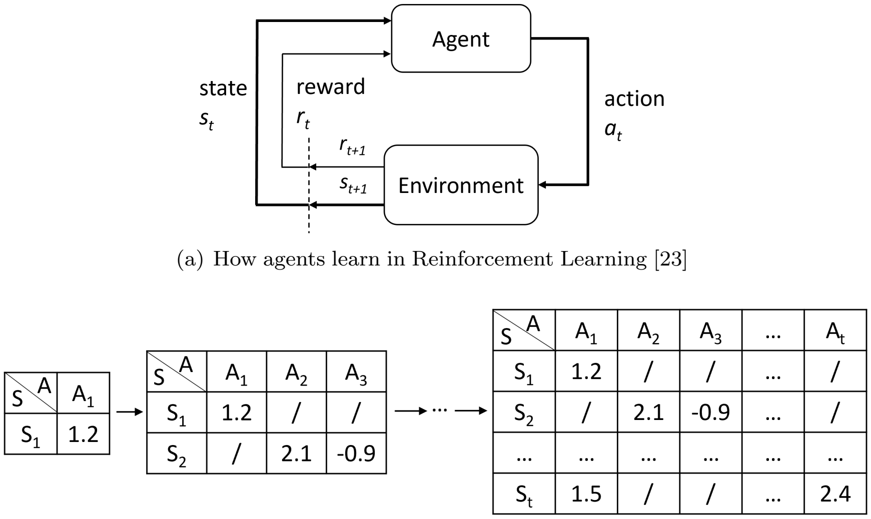
(i) **Invariance**: $E\texttt{[]}\ p$ meaning that there exists a run where all the states satisfy p, or $A\texttt{[]}\ p$ meaning that for all runs, p is satisfied by all states in each run,

(ii) **Liveness**: $A\texttt{<>}\ p$ ($A\texttt{<>}_{\leq t}\ p$) meaning that for all runs, p is satisfied by at least one state in each run (within `t` time units).

2.4. Reinforcement Learning

Reinforcement learning (RL) [23] 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. Fig. 2 depicts how an agent learns in RL. Without losing generosity, we assume that the agent starts from state S_1 and takes action A_1. When taking an action at the current state, the agent obtains the feedback of the action from the environment, including the immediate reward and the next state that agent is going to transfer to (Fig. 2(a)). Then, the agent calculates the reward of the current state-action pair and stores it in a score table. When the agent reaches the goal state, or fails, or exceeds the time limit, one round of learning is accomplished, and the score table is populated with the explored state-action pairs and their corresponding rewards (Fig. 2(b)).

Based on the types of environment, RL algorithms can be categorized into model-free RL and model-based RL. 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



(a) How agents learn in Reinforcement Learning [23]

S \ A	A$_1$
S$_1$	1.2

→

S \ A	A$_1$	A$_2$	A$_3$
S$_1$	1.2	/	/
S$_2$	/	2.1	-0.9

→ ... →

S \ A	A$_1$	A$_2$	A$_3$	...	A$_t$
S$_1$	1.2	/	/	...	/
S$_2$	/	2.1	-0.9	...	/
...	...	...	...	...	...
S$_t$	1.5	/	/	...	2.4

(b) An example of how a score table is populated in Reinforcement Learning. S stands for state, A stands for action, and / means an action is not allowed at a particular state.

Figure 2: A process of Reinforcement Learning.

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 [24] is one of the model-free algorithms, in which scores of state-action pairs are calculated by a *Q function* that satisfies the Bellman optimality equation:

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

where $q^*(s,a)$ represents the expected reward of executing action a at state s, $\mathbb{E}$ denotes the expected value function, $R(s,a)$ is the reward obtained by taking the action a at state s, $\gamma \in [0,1]$ is a discounting value indicating how much the future reward is evaluated in the calculation of the expected value of the current reward, 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'). When γ is zero, the future reward is not considered at all, which means the learning algorithm becomes a greedy algorithm that only considers the available actions at the current state. When

γ is larger than zero, the future rewards are taken into consideration. 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, no matter how the environment reacts, the MAS can finish its tasks while satisfying certain requirements, e.g., never let two agents execute a task simultaneously. The overall goal of MAS planning is:

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. *The planning problem becomes a path-finding problem only, in case the agents do not need to accomplish any tasks, but only travel to different positions in the environment. Similarly, the planning becomes task scheduling only if the agents do not need to travel, e.g., scheduling of processes in a multi-core computer system.*

Remark 2. *The requirements can be functional ones, such as task A always being started after task B, and safety ones, such as no collision with static obstacles within the environment.*

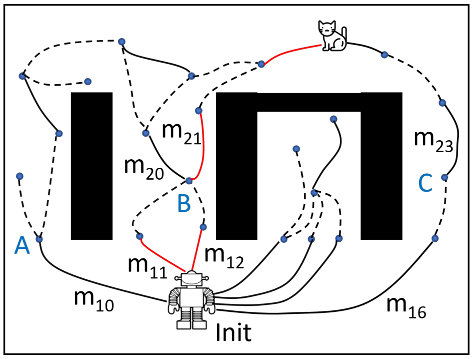
3.2. Challenges of Solving the Planning Problem

The major challenges of this problem stem from four aspects, which get amplified especially when solving the problem via algorithmic techniques. While the challenges are exposed and identified in the present study, they align with the experience reported in other literature [5, 11, 25, 14, 10, 9, 26].

- **Challenge I** (uncertainty): The agents' actions have uncertain execution time, which means agents can choose actions to perform but cannot control how much 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 [5, 11].
- **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 [14, 25].
- **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, 25].
- **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 Airborne Collision Avoidance System X (ACAS X) [26].

3.2.1. An Example of Planning for Illustrating the Challenges

To give a concrete example of large plans, in Fig. 3, we show a path-finding problem in a 2D space, where a robot tries to catch a cat. Note that our mission-planning problem combines path finding and task scheduling, which makes the model's state space to be high dimensional rather than a 2D space.



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

Limits of search-based methods. Algorithmic planning methods, such as Dijkstra's algorithm for path finding [27], and the symbolic on-the-fly algorithm for solving timed games [18], 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 [18]. However, the algorithmic methods are not scalable because they fail to solve the problem in a reasonable time when the model's state space becomes large [13].

Limits of reinforcement-learning-based methods. 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. 3(a), a path plan synthesized by reinforcement learning contains multiple routes from the robot to the cat, which results in a score table shown in Fig. 3(b). A robot under the control of a plan always chooses the actions with the highest score at each of its states. For example, if a robot is controlled by the plan in Fig. 3(b), it non-deterministically chooses among actions $\mathtt{m}_{11}$ and $\mathtt{m}_{12}$ at its initial position, because they have the highest score at state Init. However, one cannot neglect other actions at state Init before the learning finishes because score tables are populated gradually during the course of learning while the scores converge to the optimal values at the limit (see Fig. 2(b)). This causes the final score table to contain many useless data that are not optimal or even violate the requirements. Another example of useless data is the pair ($\mathtt{C}$, $\mathtt{m}_{23}$). 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, there is no guarantee on the correctness of the learning results, that is, even the actions with the highest score are not guaranteed to lead the agents towards the goal state and satisfy all requirements. This drawback of Q-learning [24] stems from three facts: i) although the scores of state-action pairs converge to the optimal values at the limit, it is unknown when they converge. Therefore, it is hard to know if the learning is sufficient enough for synthesizing a comprehensive score table that covers all the possible states; ii) the learning algorithm uses data sampled from random simulation, thus even if the scores converge, the resulting plan is not guaranteed to provide a safe policy as rare events might not be encountered during the simulation, even with very high sampling budget [28]; iii) when constructing the reward function for Q-learning, that is, the function for calculating immediate rewards, one often needs to make a trade-off between safety and performance. This would inadvertently cause the resulting score tables to accept risks of failure, as long as the performance improvement compensates the cost of failure. Hence, a post-verification on the plans synthesized by reinforcement learning is important for ensuring the correctness of the results.

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. 4, 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. 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):

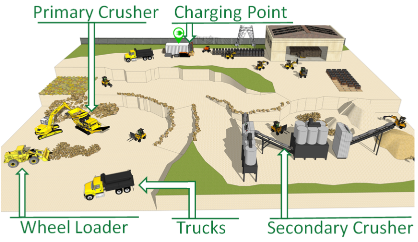


Figure 4: An autonomous quarry

- 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[4] [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 cannot be compressed either. Next, we introduce MoCReL in detail.

4. Strategy Synthesis, Verification and Compression

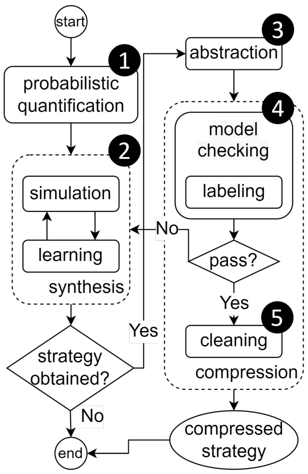


Figure 5: Workflow of MoCReL

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

The workflow of MoCReL is shown in Fig. 5.

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).

[4] Note that the mission plan is for all the agents that collaborate to accomplish a common goal. Hence, the synthesis must be done over the composed model of all agents, which dramatically increases the complexity.

Step 2: Strategy synthesis takes place, which employs the Monte-Carlo simulation in UPPAAL STRATEGO [15] 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 [29], and implement MoCReL as an external library[5].

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 synthesized strategy, which is stored in the external library of MoCReL, about the preferred actions at a given state. The preferred state-action pairs are labeled as "visited".

Step 5: If the verification fails, we go back to *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 strategy is cleaned by removing the unlabeled pairs, which completes the phase of strategy compression.

Note that increasing iteration limit is considered a valid approach only when the current learning rounds are insufficient, that is, when the counterexample returned from the model checker shows that the synthesized strategy has not covered all the states that the agent model may encounter on its way towards the goal. Besides increasing the iteration limit, one can also use the counterexamples to guide the following rounds of learning. The authors are working on this method and leave it as a future work to report. If the counterexample shows that the violation of a requirement is caused by a controllable action suggested by the synthesized strategy, one should stop increasing the iteration limit and examine the model or reward function.

Models and strategies throughout the workflow are interpreted semantically as shown later in Section 4.4. UPPAAL STRATEGO supports both the algorith-

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

mic synthesis in Uppaal TiGa [30] and the learning-based synthesis that uses reinforcement learning [20]. 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 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. We describe this feature throughout the following introduction of the TG templates.

(i) **Movement TG**: The TG template of movement models an 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 need to model the concrete trajectories, but the traveling targets and the time duration. For example, the trajectories depicted by Fig. 6 can be modeled by the movement TG, regardless of how the agents move, such as continuously in Fig. 6(a) or discretely in Fig. 6(b) and Fig. 6(c).

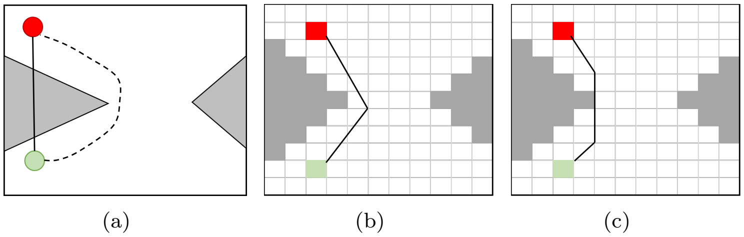
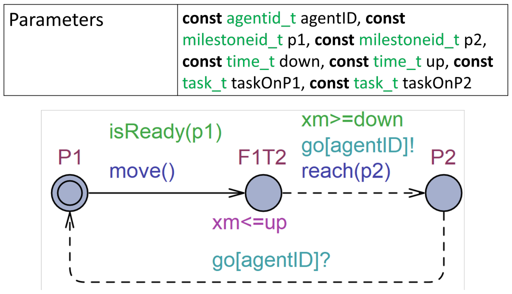


Figure 6: Examples of trajectories.



Parameters	**const** agentid_t agentID, **const** milestoneid_t p1, **const** milestoneid_t p2, **const** time_t down, **const** time_t up, **const** task_t taskOnP1, **const** task_t taskOnP2

Figure 7: The TG template of agent movement

Fig. 7 shows the template of *movement TG*, in which locations `P1` and `P2` represent any legal positions in the map. The parameters $p1$ and $p2$ are the concrete positions that locations `P1` and `P2` represent, respectively. 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 ($\texttt{xm} \leq \texttt{up}$) and guard ($\texttt{xm} \geq \texttt{down}$) regulate that the traveling time must be within the interval between `down` and `up`. The time interval is also parameterized by the parameters *down* and *up* of this template.

Now let us argue why the new movement TG template is easier to adapt than the movement template of MCRL [13]. In MCRL's movement template, all the positions that need to be visited by the agents are modeled as locations in one movement template, which means that whenever the topology of the environment changes, such as more positions are available, the movement template needs to be changed accordingly. The modification is error-prone and time consuming, especially when the number of positions is large. When using the new movement TG template, one only needs to pass different values of the parameters when the topology changes, or instantiate more models when more positions are available for the agents. For example, when the modeling granularity of movement changes from Fig. 6(b) to Fig. 6(c), one only needs to instantiate three TG models instead of two, whereas in the movement template of MCRL, one needs to add more locations and edges into the template, which is error-prone and hard to automate.

Note that the agent's current position and current status of task execution are stored in arrays *gp* and *gt*, respectively, which are shared with all the TG of movement, task execution, and monitor. The arrays *gp* and *gt* are used in the guard function `isReady` and update functions `move` and `reach`. The update functions are straightforward, which change the value of $gp[agentID]$ according to the values of the parameters $p1$ and $p2$. The guard function `isReady` is more complex, and is depicted in Algorithm 1.

Algorithm 1: isReady in the movement TG template

```
1 isReady(int position)
2 if $timeUp$ || $isGameWon()$ || $isMoniterStop(agentID)$ then
3     return $false$
4 if $gp[agentID] == position$ && $gt[agentID] == TASK\_IDLE$ then
5     if $gs[taskOnP1] == FINISHED$ || $gs[taskOnP2] == UNSTARTED$
       then
6         return $true$
7 return $false$;
```

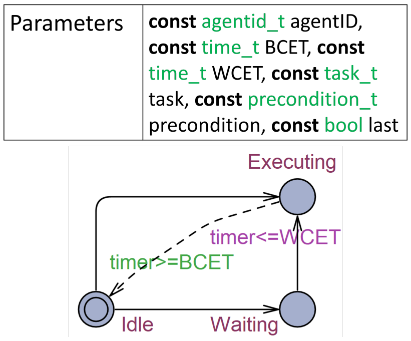
Line 2 means that if the simulation time is consumed (*timeUp* is *true*), or the mission goal is achieved (*isGameWon()* returns *true*), or the monitor has stopped (reaching the location `Stop` in Fig. 9), the agent is not allowed to move, and thus the `isReady` function returns *false*. Line 4 means that if the agent's current position is at *p1* and the agent is not running any task, it can be allowed to move. Line 5 means that only when the agent's task at *p1* is finished and the task at *p2* is not started, the movement from *p1* to *p2* is permitted. This condition is for eliminating meaningless movements among positions where tasks have been finished already.

Combining with Fig. 7, we know that the argument of *position* at line 4 is always *p1*, because one instance of the movement TG template only models one direction of the movement, that is, from *p1* to *p2*. For example, when modeling a car moving from the green point to the red point in Fig. 6(a), one needs to instantiate two models, i.e.,

$$\texttt{green2red} = \texttt{Movement(}CAR\texttt{, } GREEN\texttt{, } RED\texttt{, } 60\texttt{, } 65\texttt{, } JOB\_A\texttt{, } JOB\_B\texttt{)} \tag{3}$$

$$\texttt{red2green} = \texttt{Movement(}CAR\texttt{, } RED\texttt{, } GREEN\texttt{, } 60\texttt{, } 65\texttt{, } JOB\_B\texttt{, } JOB\_A\texttt{)} \tag{4}$$

where CAR, $GREEN$, RED, JOB_A, and JOB_B are constant integers representing the corresponding elements in the environment, respectively. When the agent leaves the green point, model `green2red` leaves location `P1`, and then finally reaches location `P2` meaning that the agent has arrived at the red point. Similarly, when the agent goes back to the green point, model `red2green` leaves location `P1` while model `green2red` goes back to its location `P1` synchronously via the channel `go[agentID]`, because `green2red` needs to be ready for the future movement from the green point to the red point.



Parameters	**const** agentid_t agentID, **const** time_t BCET, **const** time_t WCET, **const** task_t task, **const** precondition_t precondition, **const** bool last

Figure 8: TG template of task execution. The guards and functions are not shown.

(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, respec-

Table 1: An example of the guard on the edge from locations `Idle` to `Executing` in Fig. 8. The guard is a CNF formula: $C1$ && $C2$ && $C3$ && $C4$ && $C5$.

C1	!isBusy(agentID, task)
C2	isExecutable(agentID, task)
C3	precondition[agentID][task] = FINISHED
C4	task_status[task] = FINISHED
C5	!isMonitorAlert(agentID)

tively. However, the structure of the templates is the same (Fig. 8)[6]. When the agent is allowed to execute a task, the guard on the edge from location `Idle` to `Executing` is *true*. The guard is in the conjunctive normal form (CNF). Table 1 shows examples of such guards. C1 checks if the device that the task requires is busy or not; C2 checks if the agent is allowed to start a task, which is similar to the condition of line 2 in Algorithm 1; C3 checks if the task's precondition is *true*, such as the preceding tasks are finished; C4 checks if the target task that is about to be executed finishes or not; C5 checks if a monitor is alerting or not. The guard varies as the type of task changes, e.g., when 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. Note that we explicitly model the *waiting* action in this template rather than using the delay at location `Idle` because on the edge from locations `Idle` to `Waiting`, the agent's status is changed from from *TASK_IDLE* to *WAITING*. The agent's status is also used in the movement template for preventing the agent from moving when its status is not *TASK_IDLE* (line 4 in Algorithm 1). When the task is being executed, the TG *can* leave location `Executing` after the timer exceeds the BCET, and *must* leave the location when it reaches the WCET, meaning that the execution time of the task is between BCET and WCET.

(iii) **Monitor TG**: A monitor monitors a signal, e.g., the fuel level of an agent,

[6] The full templates can be found: https://github.com/rgu01/MoCReL-Experiments.git

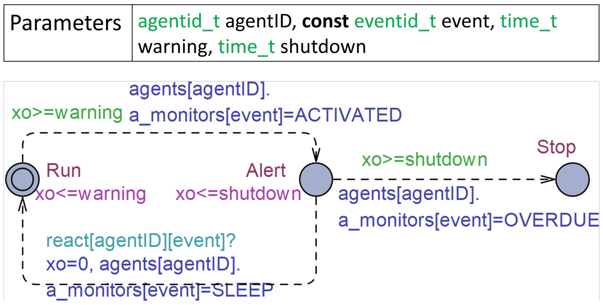


Figure 9: TG template of monitors

and triggers an event when the signal exceeds a 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 threshold. In Fig. 9, 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 (i.e., `agents[agentID].a_monitors[event]=ACTIVATED`). The corresponding task execution TG (Fig. 8) 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 (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. Parameters, such as `event` for event ID, `warning` for the threshold of time before transferring to location `Alert`, and `shutdown` for the threshold of time before shutting down the entire system, are configurable so that the monitor TG can be applied to monitor various signals in different applications.

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 (Timed)

Computation Tree Logic ((T)CTL) [16] 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. Note that one can also use $A{<>}_{\leq t}\ p$ to express p will always be eventually satisfied within t time units. For simplicity, we use $A{<>}\ p$ to define the planning problem in this paper. 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 [24]) 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 such strategies, in this section, 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 discrete variables are enough to identify the MAS's states, strategies can ignore all the clocks in the model, 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 only needs to 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 (5) in Section 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 to exhaustively verify TG under the control of strategies in Uppaal Stratego, which are synthesized via learning. Tomita *et al.* [31] divide specifications into *must specifications* that must not be violated and *desirable specifications* that may be inevitably violated. Despite the restriction, the properties of our strategies fall into the *must specifications*, such as never collide with static obstacles in the environment. Therefore, it is important to support exhaustive verification on our synthesized strategies. Note that though our strategies do not contain clocks, one can still verify the strategies against timing properties, such as always finishing all the tasks within two hours. To satisfy such timing properties, one needs to wisely design the reward function so that a strategy without clocks can take time limit into consideration. The introduction of reward functions is in Section 4.4.3.

Now, we formally 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 $\mathcal{G}$ with a score table $\mathcal{T}$ of state-action pairs is a function $\sigma : q \rightarrow \mathcal{A} \subseteq \mathcal{A}^q_{\mathcal{G}}$, 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 controllable actions that* *are allowed by $\mathcal{G}$ at state q. Let $\|\mathcal{A}\|$ be the cardinality of $\mathcal{A}$, $max(\mathcal{A}^q_{\mathcal{G}}, \mathcal{T})$ be a function that searches $\mathcal{T}$ and returns a set of actions with the highest score among actions in $\mathcal{A}^q_{\mathcal{G}}$, σ must hold the following conditions:*

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

2. *if $\|max(\mathcal{A}^q_{\mathcal{G}}, \mathcal{T})\| \geq 1$, then $\mathcal{A} = max(\mathcal{A}^q_{\mathcal{G}}, \mathcal{T})$.*

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

Unlike the strategies synthesized by search-based 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.

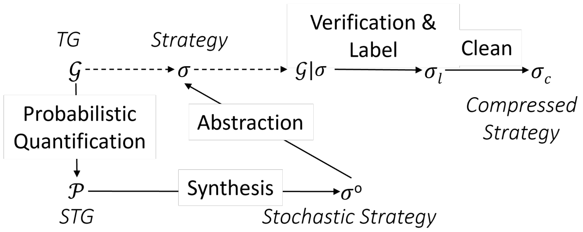


Figure 10: Model relations in the process of MoCReL

Fig. 10 shows the transformation of the model's semantics in the workflow of MoCReL. As the transformation is on the semantic level, the model's templates do not need to be changed. We elaborate on this in the following paragraphs. 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. 8, the execution time of tasks is bounded, so the non-deterministic choice of when to leave location `Executing` is transformed to a uniformly-distributed one.

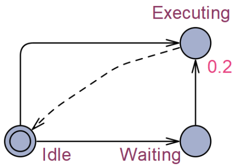


Figure 11: A task execution STG of a task with an unbounded execution time. This model does not exist in our MAS STG.

If the execution time of a task is unbounded (Fig. 11), an exponential probability distribution must be assigned to the unbounded delay on location `Executing`, where an exponential rate is used, e.g., `0.2`. However, models like the one in Fig. 11 do not exist in our MAS STG, because tasks in our problem have BCET and WCET. If a location has no invariant, which means unbounded delay is allowed there, but only outgoing controllable actions are connected to that location, such as `Idle` in Fig. 8, we still adopt uniform distribution at that location because strategies are non-lazy, meaning that agents urgently decide on a controllable action when it is available, or delay until the environment reacts. In this case, the controllable actions and delay are equally likely to be chosen, so the model templates do not need to be changed either. 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. 10), 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 non-deterministic 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 the outcome of σ°, since the former may exhibit behaviors that are highly unlikely or even do not exist in the latter. Therefore, the exhaustive 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        (5)

strategy policy = maxE(x)[<=T]{dv}-->{cv}:<> P        (6)
```

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 [23].

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 [16].

The formula "`<> P`" is a TCTL 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 "$\texttt{<> time} \geq \texttt{C}$", where `time` is a global clock in the model that is never reset and $\texttt{C} \in [0,\ \texttt{T}]$ 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 immediate rewards or penalties, respectively.

When running Query (5) 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. 3, to show 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:

$$\texttt{x} = \texttt{time} - \texttt{caught} \times \texttt{REWARD} \tag{7}$$

The variable `time` is the aforementioned global clock, `caught` is a binary integer (i.e, 1/0) indicating whether or not the cat is caught by the robot, 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.

Mistake 1: misuse of synthesis queries. If one adopts the reward function of equation 7 but mistakenly uses Query (6) 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 most 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.

Mistake 2: inappropriate reward functions. Even if one uses the query correctly, the resulting mission plan may not be able to let the robot catch the cat before the latter escapes, that is, within `N` minutes. This is because the reward function (7) does not consider the time limit. One can improve the reward function to be equation (8).

$$\texttt{x} = \texttt{time} - \texttt{caught} \ \times \ (\texttt{time<=N}) \times \texttt{REWARD} \tag{8}$$

Now, only when the robot catch the cat within `N` minutes, it is given the reward. This examples show a misuse of synthesis queries and an inappropriate design of reward functions. Even if one avoids such two types of mistakes, 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 [15]:

$$\texttt{A<>}\ \phi\ \texttt{under}\ \sigma \tag{9}$$

$$\texttt{Pr[<=T]}\ \phi\ \texttt{under}\ \sigma \tag{10}$$

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 (9) returns an absolute answer of true or false to the question of whether ϕ is always eventually satisfied, whereas Query (10) returns the probability of satisfying ϕ.

In this paper, we extend Uppaal Stratego to support Query (9) on strategies that are synthesized by learning. The pseudo-code of executing Query (9) is in Algorithm 2. In Appendix A.2, we illustrate the execution of the algorithm with an example. For the sake of readability, we overview the algorithm briefly here. To verify a liveness property like Query (9), one needs to explore the model's state space until either getting a counter-example violating the property, or until reaching all the states. Specifically, a counter-example of a liveness property like Query (9) must be either a loop, or a maximum run ending at an unbounded state where time increases indefinitely, or a deadlock, in which none of the states satisfy ϕ. Hence, once such a run is found, the verification terminates with a negative result (line 15 and line 17 in Algorithm 2).

Algorithm 2 is based on the algorithm for checking liveness properties in the literature [32]. The main difference between these two algorithms is that the state space exploration in Algorithm 2 is guided by a strategy. Specifically, when the model checker faces controllable actions (i.e., $\mathit{isControllable}(\xRightarrow{a})$ in line 22), or a delay (line 9), it calls function `Allow` to lookup the score table of a strategy and chooses the actions with the highest score. In this way, the liveness verification is guided by a strategy. In addition, the correctness and termination of Algorithm 2 are guaranteed by the algorithm for checking liveness properties in the literature [32].

Example. We show several queries that can be used in the verification of the synthesized strategy in the path-finding problem of Section 3.2, Fig. 3.

$$\begin{aligned}\texttt{strategy policy = minE(time - caught} \times \texttt{REWARD)[<=100]}\\ \texttt{\{robot.location\}-->\{\}:<> time >= 90}\end{aligned} \tag{11}$$

$$\texttt{A<> caught under policy} \tag{12}$$

$$\texttt{A[] !collide() under policy} \tag{13}$$

Query (11) synthesizes a strategy named `policy`, which allows the robot to catch the cat within 100 time units (`[<=100]`). The condition at the end of the query, i.e., `time >= 90`, specifies when to sample runs. In this example, `time` is a global clock that is never reset, and we want to sample runs that execute at least 90 time units because this is the estimated least time for the robot to catch the cat. The concrete value 90 can be replaced by any positive integer

Algorithm 2: Algorithm of liveness verification (adapted from Fig. 3 in the literature [32]): model checking $\mathcal{G} \mid \sigma$ against Query (9)

```
Function Liveness($\mathcal{G}$, $\sigma$, $\phi$):
    $ST := \emptyset$ $SD := \emptyset$ $Passed := \emptyset$
    Delay($\mathcal{G}.S_0, \neg\phi$)
    for $S_d \in SD$ do
        Search($S_d, \neg\phi$)
    return ($true$)
Function Delay($S, \varphi$):
    for $S' : S \overset{d}{\Longrightarrow} S'$ do
        if $Allow(\sigma, \overset{d}{\Longrightarrow})$ then
            if $(S' \notin SD) \wedge (S' \models I(S.l) \wedge \varphi)$ then
                $push(SD, S')$
Function Search($S, \varphi$):
    $S := S \wedge \varphi$
    if $S \neq empty$ then
        if $loop(S,ST)$ then
            exit($false$)                                          // Loop found
        if $unbounded(S) \vee deadlocked(S)$ then
            exit($false$)                                          // Maximal run found
        $push(ST, S)$
        if $\forall S' \in Passed : S \nsubseteq S'$ then
            for $S_a : S \overset{a}{\Longrightarrow} S_a$ do
                // If action a is uncontrollable or allowed, it can be chosen.
                if $\neg isControllable(\overset{a}{\Longrightarrow}) \vee Allow(\sigma, \overset{a}{\Longrightarrow})$ then
                    Delay($S_a$, $\varphi$)
                    for $S_d \in SD$ do
                        Search($S_d$, $\varphi$)                   // Recursive all
    $Passed := Passed \cap \{pop(ST)\}$                            // Move from stack to Passed
Function Allow($S, action$):
    if $NumControllable(S) == 1$ then
        return ($true$)
    if $action \in best(\sigma, S)$ then
        $label(\sigma, S, action)$                                 // Label (S, action) in $\sigma$
        return ($true$)
    else
        return (false)
```

depending on the designer's experience. Normally, it is less than or equal to the total simulation time, i.e., 100 in this example, because no clock in the model can exceed the total simulation time. Query (12) verifies the robot model under the control of `policy` to see if it can always eventually catch the cat. Query (13) 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 whether collisions between the robot and obstacles never happen.

Besides the possible errors in the resulting strategies, as presented in the path-finding example, strategies can be memory consuming for containing too much useless data. With the help of the external library where MoCReL is implemented, we can leverage queries in the form of Query (9) 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 can enquire 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 of actions, we also label the state-action pairs that have the highest score as "selected" because they are selected and verified by the model checker.

When verifying a liveness property (e.g., Query (9)), 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 (9)) is always eventually *true*. Therefore, if the liveness property is satisfied, the exhaustive model checking guarantees the labelled state-action pairs to eventually reach the states where the property is true regardless of which and when the uncontrollable actions are taken by the environment. 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 pairs (*cleaning* in Fig. 5). By verifying the compressed strategy again, we can see if the new strategy preserves the liveness property that is met by the original strategy.

Algorithm 3: MoCReL algorithm

```
1  Function Main($\mathcal{G}$, $\mathcal{Q}$, $iterationNum$, $totalNum$, $goodNum$, $formula$):
2      Strategy $\sigma := \emptyset$, $\sigma_c := \emptyset$
3      Stochastic Strategy $\sigma^\circ := \emptyset$
4      STG $\mathcal{P} := ProbabilisticQuantification(\mathcal{G})$
5      while $\neg Liveness(\mathcal{G},\ \sigma,\ \mathcal{Q})$ do
6          $\sigma^\circ := Learn(\mathcal{P}, iterationNum, totalNum, goodNum, formula)$
7          $\sigma := Abstraction(\sigma^\circ)$
8          $Update(iterationNum,\ totalNum,\ goodNum)$
9      $\sigma_c := Clean(\sigma)$
10     return $(\sigma_c)$
```

4.4.6. Soundness of MoCReL

Algorithm 3 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 4 in Appendix A.1). The function `Liveness(`$\mathcal{G}, \sigma, \mathcal{Q}$`)` at line 5 runs Algorithm 2, 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, as score tables are empty initially and populated on the fly with the visited state-action pairs. Line 9 cleans strategy σ by removing the unlabeled data, thereby compressing the strategy.

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 3 terminates and returns a strategy* σ_c*, then* $\mathcal{G} \mid \sigma_c \models \mathcal{Q}$*.*

Proof. Obviously, Algorithm 3 terminates with two cases:

1. `Liveness(`$\mathcal{G}$, σ, $\mathcal{Q}$`)` returns *true* (line 5 in Algorithm 3), when Algorithm 3 will eventually return σ_c (line 10 in Algorithm 3);

2. `Liveness(`$\mathcal{G}$, σ, $\mathcal{Q}$`)` has a negative result (`exit` at line 16 and line 18 in Algorithm 2), and causes Algorithm 3 to terminate with no strategy returned (line 10 in Algorithm 3 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 in the rest of the proof all refer to Algorithm 2):

(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 of the states satisfy $\neg\phi$ or none of the states 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 where time can increase indefinitely, 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 3 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 [24]. 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. 12 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.

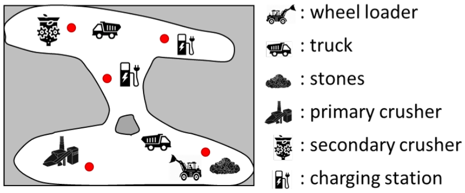


Figure 12: An autonomous quarry

In the experiments, we aim to synthesize mission plans that indicate the agents which milestones to go or which tasks to execute. Therefore, we choose a coarse granularity of agent movement, that is, between every pair of milestones. The traveling times among milestones are calculated by using the A* algorithm [3]. Now the state space of the model is mostly influenced by the number of agents, as the number of states in the composed model of all agents 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 whether 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[7].

According to our previous study, the method that purely uses search-based algorithms (namely TAMAA) 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]. A learning-based method (namely MCRL) can deal with more agents and flexible task execution time, but collaborations and events are not supported [12]. The experiments in this section include collaborations among agents and a battery-low event. Note that collaborations mean agents carry out a common task at the same milestone simultaneously, such as a wheel loader unloading stones into a truck. Maps in the experiments are also complex, i.e., some models contain 2-4 primary crushers and 1-2 secondary crushers. Table 2 shows the ranges of parameters scoping the problems that TAMAA, MCRL, and MoCReL can solve, respectively. A detailed comparison between the search-based method and learning-based method is reported in the literature [13].

Table 2: Problems that TAMAA, MCRL, and MoCReL can solve, respectively

Methods	Agent amount	Task amount	Existence of events	Task types
TAMAA [9]	2 - 4	3	Yes	2
MCRL [13]	2 - 6	3	No	2
MoCReL	2 - 6	3 - 6	Yes	4

[7] Discrete variables are in the queries of models at https://github.com/rgu01/MoCReL-Experiments.git

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 3 that we introduce in the next section.

Table 3: Results of strategy synthesis, verification, and compression. Abbreviations: 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 against the compressed strategy (VER).

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 first series of experiments is conducted on the full set of models while the second is restricted to a subset. The set of models is grouped intro three categories:

- *Category I*: No charging, no choice in crusher, large numbers of agents up to 6, a small number of crushers (2), and a fixed medium value of the trucks' capabilities (20).
- *Category II*: No charging, choice in crusher, medium numbers of agents

(2 - 5), large numbers of crushers (3 - 6), and a range of the trucks' capabilities (10 - 30).

- *Category III*: Charging, no choice in crusher, small numbers of agents (2 - 3) and crushers (2), and a fixed large value of the trucks' capabilities (50).

The second series of experiments is conducted on a model `game6-B` in Table3 and its two variants that change the capability of trucks (CAP), that is, 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 (10). 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. In both series of experiments, the target amount of stones to be transported is the same in all models.

5.3. Experiment Results

In Table 3, Column "VER" shows the results of verifying the compressed strategies against queries in the form of Query (9). Column "RUNS" includes the numbers of runs that needs to be sampled for synthesizing a valid strategy, which are picked empirically.

Synthesis time. In category I, the time of synthesizing strategies is relatively short. 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[8]. 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 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.

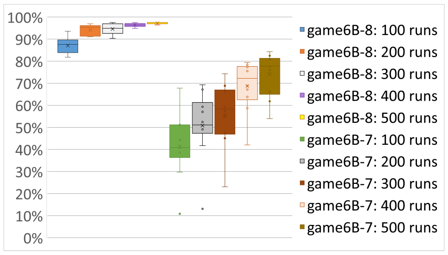


Figure 13: Distribution of mean probabilities of satisfaction over 10 experiments

Learning efficiency. Fig. 13 shows the mean probabilities of agents reaching their goal (i.e., satisfying Query (10)). 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[9]) are the largest. This indicates

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

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

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 size reductions of compressed strategies are up 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 3. 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[10]. 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 size reductions 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 numbers of milestones and tasks are 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.

[10] See the strategies of `game4-A` at https://github.com/rgu01/MoCReL-Experiments.git.

6. Discussion

Although the experiments demonstrate MoCReL's good performance of strategy synthesis and compression, the approach has a few assumptions and limitations that are discussed in the following.

Continuous variables are not included in strategies. MoCReL observes a partial state space of an agent model, which only covers discrete variables. It is not trivial to include continuous variables, e.g., time, in strategies while preserving the exhaustive model checking, because the score tables will be infinitely long due to the infinite values of the continuous variables. One may adopt techniques of representing sets of states symbolically, as zones [16], and transform the infinite state space into a finite state space. We leave this as future work.

Low learning efficiency in some cases. As shown in the experiments, when reaching the goal becomes a rare event for the agents, the learning efficiency of reinforcement learning drops dramatically. Rare events simulation is a longstanding problem with simulation technology [33, 34]. Techniques such as importance sampling [35] have been investigated for solving this problem [36]. We believe that, by exploiting counterexamples, one can also increase the learning efficiency of reinforcement learning algorithms.

7. Related Work

Synthesis of strategies for MAS has been an increasingly researched area in recent years. Formal methods have been adopted to complement planning algorithms with correctness guarantees. Alur *et al.* [37] use game theory for compositional synthesis of reactive controllers from LTL specifications for multi-agent systems, in which agents can be controllable or uncontrollable. Their method assumes the LTL specifications can be separated into several sub-specifications that concern subsets of agents, respectively. In our problem, agents are designed to accomplish a common goal and the requirements concern all of them. Křetínskỳ [38] investigated the combination of LTL, Steady-State Policy Synthesis (SSPS), and long-run average reward (LRA) on synthesizing policies that resolve Markov decision processes (MDP). However, our planning problem concerns properties expressed by (T)CTL, and our method includes strategy synthesis and compression. Gleirscher *et al.* [39] introduce an approach for synthesis and verification of safety controllers for human-robot collaboration. Their synthesis means selecting a safe controller over several models that are created by control engineers according to different applications. Our synthesis does not need engineers to manually design controllers as it is over agents' motions, such as movement and task execution, whose model templates are defined already.

In the field of strategy synthesis and verification, Uppaal and its branches have been employed in many studies. Andersen *et al.* [40] 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. Basile et al. [41] use Uppaal Stratego to solve the strategy synthesis problem for autonomous driving in a moving block railway system. The authors demonstrate the applicability of Uppaal Stratego in a concrete case study. They model the railway system as a stochastic priced timed game and thus apply statistical model checking on their resulting (safe) strategy. Our verification is exhaustive, which aims to see if the agents can achieve their goal safely and timely, regardless of how the environment acts non-deterministically. Bersani *et al.* [14] present PuRSUE (Planner for RobotS in Uncontrollable Environments), which supports users to configure their robotic applications and automatically generate their controllers by using Uppaal TiGa. The main difference between our work and this is that the authors base their synthesis on an exhaustive search of the model's state space, which provides a correct-by-construction solution, but the scalability of their method is inherently limited.

In the field of combining formal methods with reinforcement learning (RL), Behjati *et al.* [42] attempt to solve the state-space-explosion problem of model checking LTL properties by using RL. Bouton *et al.* [43] propose a method that enforces probabilistic guarantees on agents during the course of RL. Jothimurugan *et al.* [44] propose *DIRL*, a synthesis approach that interleaves Djikstra's algorithm with RL to train agents. In comparison to the mentioned related work, the correctness guarantee provided by MoCReL is not on the course of learning or on formal specification of the reward functions and agents' tasks. Instead, MoCReL provides an exhaustive post-verification of the synthesis results, which is more scalable than verifying the original agent models. Additionally, counterexamples returned from model checking show the agents' behaviors that violate the requirement, which constitutes valuable feedback.

In the area of strategy compression, Julian *et al.* explore several ways of compressing strategies by using origami compression [45] or deep neural networks [26][46]. Ashok *et al.* propose a decision-tree-based method for concisely representing strategies [47][48]. Their tool *dtControl* is able to compress strategies produced by Uppaal TiGa. Piterman *et al.* use a method that minimizes strategies by removing redundant states [49]. Compared with these methods, the strategy compression in MoCReL focuses on removing the unused data in the strategies rather than representing them in different forms. Since it relies on exhaustive model checking, compression in MoCReL inherently provides a safety guarantee of strategies, which needs extra effort to achieve in other methods [46].

8. Conclusions and Future Work

We present a new method, namely MoCReL, for synthesis, verification, and compression 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 doing various types of tasks. 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, A* aesthetic optimizations, in: Game programming gems, Charles River Media (2000).

[4] S. M. LaValle, Rapidly-exploring random trees: A new tool for path planning, Tech. rep. (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(λ) 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, pp. 73–92.

[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] M. M. Bersani, M. Soldo, C. Menghi, P. Pelliccione, M. Rossi, PuRSUE-from specification of robotic environments to synthesis of controllers, Formal Aspects of Computing (2020).

[15] 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.

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

[17] 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).

[18] 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.

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

[20] 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.

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

[22] K. G. Larsen, P. Pettersson, W. Yi, UPPAAL in a nutshell, International journal on software tools for technology transfer 1 (1-2) (1997) 134–152.

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

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

[25] A. Atyabi, S. MahmoudZadeh, S. Nefti-Meziani, Current advancements on autonomous mission planning and management systems: An auv and uav perspective, Annual Reviews in Control 46 (2018) 196–215.

[26] 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.

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

[28] C. Jegourel, K. G. Larsen, A. Legay, M. Mikučionis, D. B. Poulsen, S. Sedwards, Importance sampling for stochastic timed automata, in: International Symposium on Dependable Software Engineering: Theories, Tools, and Applications, Springer, 2016, pp. 163–178.

[29] 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.

[30] 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, pp. 121–125.

[31] T. Tomita, A. Ueno, M. Shimakawa, S. Hagihara, N. Yonezaki, Safraless LTL synthesis considering maximal realizability, Acta Informatica 54 (7) (2017) 655–692.

[32] 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.

[33] J. Morio, M. Balesdent, D. Jacquemart, C. Vergé, A survey of rare event simulation methods for static input–output models, Simulation Modelling Practice and Theory 49 (2014) 287–304.

[34] J. Frank, S. Mannor, D. Precup, Reinforcement learning in the presence of rare events, in: Proceedings of the 25th international conference on Machine learning, 2008, pp. 336–343.

[35] P. W. Glynn, D. L. Iglehart, Importance sampling for stochastic simulations, Management science 35 (11) (1989) 1367–1392.

[36] J. P. Hanna, S. Niekum, P. Stone, Importance sampling in reinforcement learning with an estimated behavior policy, Machine Learning 110 (6) (2021) 1267–1317.

[37] R. Alur, S. Moarref, U. Topcu, Compositional synthesis of reactive controllers for multi-agent systems, in: International Conference on Computer Aided Verification, Springer, 2016, pp. 251–269.

[38] J. Křetínskỳ, LTL-constrained steady-state policy synthesis, arXiv preprint arXiv:2105.14894 (2021).

[39] 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, Science of Computer Programming 218 (2022) 102809.

[40] 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.

[41] D. Basile, M. H. ter Beek, A. Legay, Strategy synthesis for autonomous driving in a moving block railway system with uppaal stratego, in: International Conference on Formal Techniques for Distributed Objects, Components, and Systems, Springer, 2020.

[42] R. Behjati, M. Sirjani, M. N. Ahmadabadi, Bounded rational search for on-the-fly model checking of LTL properties, in: FSE, Springer, 2009, pp. 292–307.

[43] 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.

[44] K. Jothimurugan, S. Bansal, O. Bastani, R. Alur, Compositional reinforcement learning from logical specifications, Advances in Neural Information Processing Systems 34 (2021).

[45] 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.

[46] 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.

[47] 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.

[48] 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, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2021, pp. 326–345.

[49] N. Piterman, A. Pnueli, Y. Sa'ar, Synthesis of reactive (1) designs, in: International Workshop on Verification, Model Checking, and Abstract Interpretation, Springer, 2006, pp. 364–380.

Appendix A. Appendix

Appendix A.1. Algorithm of Synthesis

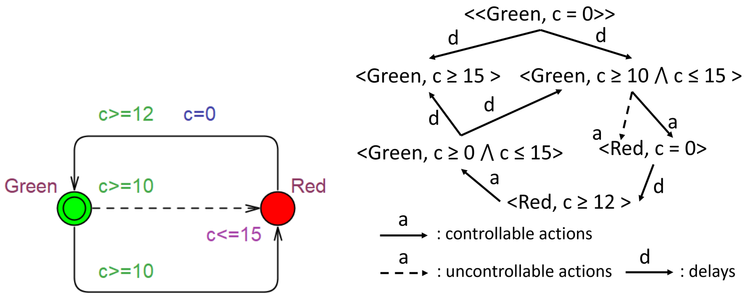
Algorithm 4 is the simplified pseudo-code of running Query (5) in Uppaal Stratego. Details of this algorithm are in the literature [13].

Algorithm 4: Simplified algorithm behind the `minE`-query (adapted from Algorithm 1 in the literature [13])

```
minE(tg, iterationNum, totalNum, goodNum, formula)
int $iterations = 0$
int $bestFitness = \infty$
Strategy $best = empty$
Strategy $aStrategy = empty$
for $iterations < iterationNum$ do
    int $totalRuns = 0$
    int $goodRuns = 0$
    for $totalRuns < totalNum$ do
        Run $aRun$ = simulate($tg$, $aStrategy$)
        if $aRun$ satisfies formula then
            $aStrategy$ = learn($aRun$)
            $goodRuns++$
            if $goodRuns \geq goodNum$ then
                break
        $totalRuns++$;
    if $goodRuns \geq goodNum$ then
        $fitness$ = evaluate($aStrategy$)
        if $fitness < bestFitness$ then
            $bestFitness = fitness$
            $best = aStrategy$
    $iterations++$
return $best$;
```

Appendix A.2. Algorithm of Verification and Labeling

In this section, we illustrate the execution of Algorithm 2 by an example in Fig. A.14. In Algorithm 2, line 3 passes the initial state S_0 of the TG $\mathcal{G}$ and the negation of the state formula of Query (9), 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 the literature [32]. In this paper, we adapt this function on symbolic states (i.e., zones) by using difference bounded matrices (DBM) in Uppaal. Fig. A.14 shows an example



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

Figure A.14: An example of a TG and its semantic model.

of a UTA modeling 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. 14(b). The function `Allow`$(\sigma, \overset{d}{\Longrightarrow})$ 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. 14(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 on 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 (9) 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 onto the stack SD for further exploration. In Fig. 14(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 on 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 [32], 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 onto ST and continues to explore the unvisited states (line 20). For example, in Fig. 14(b), both succeeding states of the initial state are pushed onto 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. 14(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. 14(a), namely trafficLight, and we want to verify a liveness property: A<> trafficLight.Red. By following Algorithm 2, 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. The TG Templates

1355 Figure A.15 depicts the TG of task execution. Figure A.16 depicts the TG of `Referee` that is used in the experiments (See Section 5).

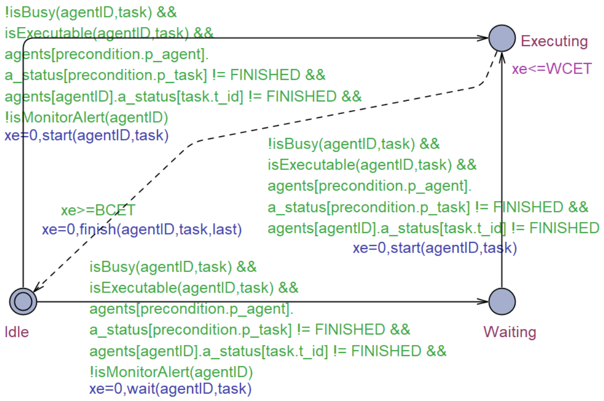

!isBusy(agentID,task) &&
isExecutable(agentID,task) &&
agents[precondition.p_agent].
a_status[precondition.p_task] != FINISHED &&
agents[agentID].a_status[task.t_id] != FINISHED &&
!isMonitorAlert(agentID)
xe=0,start(agentID,task)
Executing
xe<=WCET
!isBusy(agentID,task) &&
isExecutable(agentID,task) &&
agents[precondition.p_agent].
a_status[precondition.p_task] != FINISHED &&
agents[agentID].a_status[task.t_id] != FINISHED
xe=0,start(agentID,task)
xe>=BCET
xe=0,finish(agentID,task,last)
isBusy(agentID,task) &&
isExecutable(agentID,task) &&
agents[precondition.p_agent].
a_status[precondition.p_task] != FINISHED &&
agents[agentID].a_status[task.t_id] != FINISHED &&
!isMonitorAlert(agentID)
xe=0,wait(agentID,task)
Idle
Waiting

Figure A.15: A TG template of agent task execution

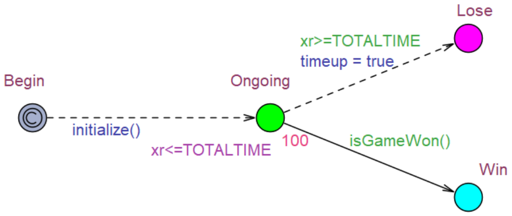


Figure A.16: The Referee TG

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: 1360 `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 (5) 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 action, 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 Section 4.4. Additionally, when under verification (Query (9) 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.