Model-Based Policy Synthesis and Test-Case Generation for Autonomous Systems

Rong Gu
Mälardalen University, Sweden
rong.gu@mdu.se

Eduard Enoiu
Mälardalen University, Sweden
eduard.paul.enoiu@mdu.se

***Abstract*—Autonomous systems are supposed to automatically plan their actions and execute the plan without human intervention. In this paper, we propose a model-based two-layer framework for policy synthesis and test-case generation for autonomous systems. At the high-level layer of the framework, we have two kinds of methods for synthesizing policies whose correctness is guaranteed by model checking. The autonomous system's controller executes synthesised policies at the low-level layer. As the kinematics of autonomous systems is often nonlinear and the environment may influence the results of their actions, formally verifying the controllers is extremely difficult. We propose a novel method for generating test cases for the controllers at the low-level layer. The method employs reinforcement learning for test-case generation and model checking to ensure that the test cases faithfully realize the execution of the policy. The framework is designed in Uppaal Stratego, which integrates model checkers and algorithms for policy synthesis. Therefore, the framework separates concerns and seamlessly interchanges the information between two layers.**

***Index Terms*—autonomous systems, model checking, testing, test-case generation**

I. Introduction

Autonomous systems, such as self-driving vehicles, are systems that operate without human intervention. They are often designed to collaborate to accomplish a mission that requires executing a sequence of tasks repetitively. Therefore, such systems need to be intelligent in the sense that they can plan their paths toward the target positions without a collision, schedule their tasks with respect to a regulated order of execution, and accomplish the mission eventually.

Path-finding algorithms such as A* [1] are often used for navigation. However, task scheduling is outside the scope of such algorithms, which requires the task executors (e.g., autonomous systems) to carry out the mission by conforming to the prerequisites of its tasks, such as task *A* must be followed by task *B*. Even if a mission plan that includes paths and task schedules is computed, the actual execution of the mission plan is error-prone because the real trajectories of executing a plan have *tracking errors*. Figure 1 depicts a planned path and a possible real trajectory tracking this planned path. First, the planned path is infeasible for any vehicle because of the sharp turning at each waypoint. Second, due to the limited mobility or inertia of vehicles, a deviation from the planned path is inevitable. The deviation is often known as *tracking errors*. Hence, we need to find an approach to check if the execution of the mission plan is correct. For brevity, we call a mission plan for a set of autonomous systems a policy henceforth.

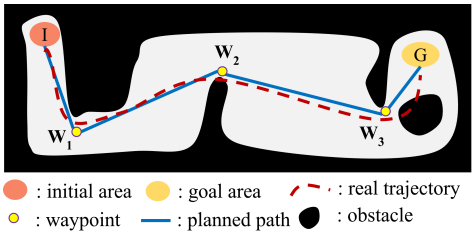


Figure 1. Examples of a planned path and a possible real trajectory

Software development of autonomous systems has to follow safety regulations and standards. These standards require rigorous testing practices [2]. Software testing is a practice to gain confidence that the autonomous system works as intended or that new faults can be discovered. In this domain, software testing often aims to find faults as early in the development process as possible in a more efficient way. Design and architectural models are created during software development and testing using these designs as input to create tests intended to check if the system meets its specifications in terms of the overall interactions among the system components. This type of testing is sometimes known as system testing [3], and its main purpose is to discover design problems. Testing, in this case, may address such properties as functional correctness, real-time guarantees and performance [4].

In this paper, we introduce our framework for policy synthesis and test-case generation for testing the correctness of policy execution. As policy synthesis and execution are inherently different and concern different aspects of autonomous systems, our framework provides a separation of concerns. Concretely, the framework consists of two layers, where a high-level layer (HL) is responsible for policy synthesis and a low-level (LL) focuses on the execution of the synthesized policies. As a continuation of our former framework for modelling and verification of autonomous systems [5], this new framework has the following improvement:

- A more comprehensive methodology for policy synthesis. Policy synthesis in the new framework considers the actions of both the systems and the environment. Methods for synthesis can be based on graph search or learning. Most importantly, policies synthesized by the HL can

be directly used in the models at the LL and control the (possibly nonlinear) model of autonomous systems to move and perform their tasks.
- An approach for automatic test-case generation is proposed for the LL. The test-case generation is based on model checking [6] and reinforcement learning [7] and is useful in experiments (Section V).

The framework is developed in UPPAAL Stratego [8], which integrates the Uppaal suite of (statistical) model checking [9] [10] and policy synthesis for timed games [11]. We use (priced) timed games ((P)TG) for policy synthesis and test-case generation [12]. At the HL of the framework, we provide a graph-search-based method [13] and a learning-based method [14] to synthesize policies that are guaranteed to accomplish the mission. At the LL of the framework, we provide a method for generating trajectories that faithfully realize the execution of policies. These generated trajectories are used as test cases for detecting errors where the deviation from the policy exceeds the boundary of tracking errors.

In all, the contributions of this paper are listed as follows:

- A framework for policy synthesis and test-case generation for autonomous systems. As far as we know, this framework is the first attempt to integrate both aspects into a holistic solution.
- An automatic test-case generation for testing the policy execution of autonomous systems. The generation is based on reinforcement learning and the test cases are guaranteed to be a faithful realization of a policy by model checking.

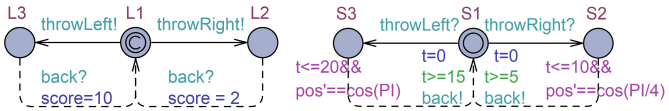
The remainder of the paper is organized as follows. Section II presents the background knowledge for reading this paper. Section III defines the key concepts of our framework and methods and presents the problem this paper solves. Section IV is a description of our framework and methods for policy synthesis and test-case generation. Section V is about the experiments for showing the applicability of our methods. We introduce the related work of this study in Section VI before presenting the conclusion and future work in Section VII.

II. Background

In this section, we introduce the background knowledge of this paper, including timed games and their policies. We also introduce a tool for solving timed games. In this paper, we denote the sets of real numbers as $\mathbb{R}$.

A. Timed Games and Uppaal Stratego

A *timed automaton* (TA) is a finite-state automaton extended with real-valued variables, called *clocks*, suitable for modelling real-time systems [15]. A *timed game* (TG) is a TA with its set of actions partitioned into *controllable* and *uncontrollable* ones. Uppaal Stratego [8] is a tool that integrates the Uppaal suite: a symbolic model checker [9], a statistical model checking [10], a symbolic policy synthesizer [16] and a statistical policy synthesizer [8] for timed games (TG). Imagine there is a game of disk throw. The goal of winning the game is to keep the disk in the air as long as possible and catch the disk when it comes back. Fig. 2 depicts two TG of such a game in Uppaal Stratego. In a



(a) A TG of a player (b) A PTG of a disk

Figure 2. An example of a network of (P)TG in Uppaal Stratego.

TG template, locations (e.g., `L2`) are blue circles. The double circles (e.g., `S1`) denote the initial location. *Clocks* (e.g., `t`) are special variables that increase simultaneously when the TG is executed. *Invariants* (e.g., `t<=10`) on locations must be *true* when the TG stays at the location. *Edges* connecting locations 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. Edges are enabled when the *guards* (e.g., `t>=5`) on them are *true*. *Assignments* on edges reset clocks (e.g., `t=0`) or update data variables (e.g., `score = 10`). When ordinary differential equations are used in TG, the changing rates of clocks are defined (e.g., `pos'==cos(PI/4)`). These TG are called priced timed games (PTG). A *network* of (P)TG (NPTG) is a parallel composition of (P)TG that can synchronize via *channels* (e.g., `back!` is synchronized with `back?`). Transitions can take place when a channel is declared as a *broadcast channel* even when no other synchronous transitions are enabled. When a location is marked as *committed* (e.g., `L1`), no delay is allowed at that location, and the next transition must start from one of the committed locations in the NPTG.

B. Policy Synthesis and Model Checking

A solution for winning a timed game is a policy that chooses controllable actions at different states of the model such that it eventually reaches a *winning* state. For example, in the disk throw game in Figure 2, players can choose the direction of throwing their disks, but when the disks return is decided by the environment (i.e., uncontrollable actions). A winning policy always chooses to throw the disk to the left because it makes the disks fly the longest time (i.e., `[15, 20]`. A formal definition of policies for timed games is in literature [11]. Uppaal Stratego provides various methods of synthesizing winning policies and linking a library to the tool as an external learning algorithm. In this paper, we utilize this function to implement our policy synthesis methods and test-case generation method.

Model checking is a technique that explores the state space of a model and checks if it satisfies properties expressed by logic formulas, such as computation tree logic (CTL) [6]. Uppaal Stratego has a model checker that we utilize in this paper to verify whether the generated test cases hold our desired features. In this paper, we use the following forms of CTL properties, where p is an atomic proposition over the locations, clocks, and data variables of TG:

(i) *Invariance*: $A[]\ p$, meaning that for all the states at all paths of the model, p is satisfied,
(ii) *Liveness*: $A<>\ p$, meaning that for all paths of the model, p is satisfied by at least one state at each path.

C. Automated Test Case Generation

If software testing is severely constrained, this typically means that less time is devoted to manually designing highly effective test cases. As a solution to this problem, automated test generation [17] has been proposed to complement manual testing and allow some test cases to be created with less effort. However, it has been a problem for software professionals and researchers over the last decades to develop effective, applicable and practically relevant test generation techniques, and tools [18]. In the case of using a model for test generation, we speak about model-based testing (MBT), which has gained a lot of attention [19]. In MBT, test artefacts are generated using an explicit model representing the environmental, functional and non-functional behaviour of the system based on a certain modelling language and notation such as Unified Modelling Language (UML), finite state machine (FSM) or timed automata, just to name a few formalisms.

III. Problem Definition

In this section, we describe the system and problem that this paper solves. Figure 3 depicts the structure of an autonomous system that works in a confined environment, where the system's data flow is also shown. From the beginning, the information of the environment (namely *mission map* or shortly *map*) is input into a module called *mission planner*, where a policy for the autonomous system is constructed.

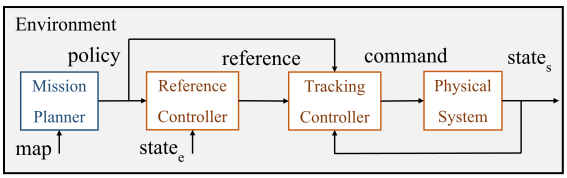


Figure 3. The structure of an autonomous system

A. Mission Map, Controller, and Policy

We define the input of the mission planner as follows:

Definition III.1 (Mission Map)**.** *A mission map is a 5-tuple* $\mathcal{M} =< \mathcal{X}, \boldsymbol{O}_u, \mathcal{I}, \mathcal{G}, Req >$*, where (i)* $\mathcal{X} \in \mathbb{R}^d$ *is the state space of the map, (ii)* $\boldsymbol{O}_u \subseteq \mathcal{X}$ *is the unsafe area, (iii)* $\mathcal{I} \subseteq \mathcal{X}$ *is the initial area, (iv)* $\mathcal{G} = \{G \mid G \subseteq \mathcal{X}\}$ *is a set of goal areas, and (v)* Req *is a set of temporal logic formulas.*

For brevity, we use *map* instead of *mission map* in the remainder of the paper. Figure 1 shows a map of 2-dimensional space (i.e., $d = 2$). In other scenarios, the dimension of the state space can be high, such as in robotic arms [20]. The safety of such robotic arms is not only about their physical positions in a 3D space but also the time points and speeds of arriving at different positions. Hence, time and speed are another two dimensions of the map. There are multiple goal areas and a set of requirements written in temporal logic in a map because autonomous systems are required to not only reach the goal areas but also fulfil the requirements of when to reach there or the order of reaching there, etc.

Given a map, a mission planner can generate a guaranteed policy to guide the autonomous system to avoid unsafe areas, reach the goal areas, and satisfy the requirements, regardless of how the environment may behave. For example, according to its schedule, a self-driving bus can decide when to leave the bus terminal and the order of arriving at the bus stations, but the travelling time is influenced by the environment, such as the weather. Our policy must tolerate such uncertainties and guarantee the system to always accomplish its mission. Policies can be in various forms, such as score tables in Q-learning [7] and artificial neural networks [21]. Regardless of its form, a *winning* policy guides autonomous systems to reach their goal. Winning policies are referred to as policies in the remainder of this paper. Note that a policy alone is not executable because it must be executed by a controller. Nevertheless, a policy can be deployed on different controllers as it has no knowledge of its controller. Additionally, policies make several assumptions, such as the time intervals of performing actions, which must reflect a correct estimation of the actual policy execution. We call the assumptions the *premises* of a policy.

B. Reference Trajectory and Real Trajectory

As depicted in Figure 3, when a reference controller executes a policy, it generates a *reference* trajectory for the system to track. As the physical system often has tracking errors, its tracking controller is designed to produce *control commands* for compensating for the system's deviation from the reference trajectory. Intuitively, a policy in Figure 1 may consist of waypoints of the planned path and actions of moving to different directions at each of the waypoints. A corresponding reference trajectory is the piece-wise continuous path denoted by the blue line, and a real trajectory tracking this reference trajectory is the dotted red line.

As aforementioned, autonomous systems can only partially control the results of their actions because the environment can also impact the results. Therefore, in Figure 3, we use two kinds of states, namely $state_s$ (states of the autonomous system) and $state_e$ (states of the environment). The former consists of variables controlled by the autonomous system, such as the system's speed. The latter consists of variables (partially) controlled by the environment, such as the time of finishing actions performed by autonomous systems. As Defintion III.1 defines the stationary elements in an environment, $state_e$ reflects its dynamic states. For example, when a robot picks up a box and puts it on a conveyor belt, the $state_s$ refers to its state, which changes when the robot's actions are done. However, when to change $state_e$, i.e., when the box is delivered, is decided by the environment, as the speed of the conveyor belt may vary. If there is another robot waiting for the box at the other end of the conveyor belt, its

policy must consider the varying time of delivery. Hence, when executing a policy, a reference controller must perceive $state_e$, read the policy, and generate a reference trajectory accordingly to accomplish the mission no matter how the environment behaves. Formally, a trajectory in a map $\mathcal{M}$ over a duration D is a function $\psi : [0, D] \to \mathcal{X}$, which maps each time $t \in [0, D]$ to a state $x \in \mathcal{X}$.

As policies neglect the details of action execution, reference trajectories are supposed to fill in such details. For example, a policy may not have information on the velocity at which an autonomous system should travel. When a reference controller executes the policy, it needs to fill in the system's velocities at different positions such that it can catch the deadline of the mission and fully stop when reaching the destination. In summary, reference trajectories have four features:

1) Reference trajectories must comply with the physical capability of the system, e.g., the highest speed.
2) Reference trajectories must hold the premises of the policy, e.g., time intervals of executing actions.
3) Reference trajectories must *realize* the policy *faithfully*.
4) Reference trajectories must satisfy Proposition III.1.

The first two features are straightforward because the details of the action that reference trajectories fill in must be realistic (i.e., compliant with the system's capability) and compatible with the policy (i.e., holding the policy's premises). The third feature means that the reference trajectories must not contain behaviours not allowed by the policy (i.e., faithfulness) and miss any behaviours that the policy presents (i.e., realization). The fourth feature is about the validity of reference trajectories. Specifically, we want the reference trajectories to avoid unsafe areas and visit the goal areas in a manner that satisfies the mission's requirements. Moreover, to ensure the validity of the real trajectories, reference trajectories must tolerate errors, that is, the differences between the real trajectories and reference trajectories. Formally, we define the reach-avoid property that we want the trajectories to hold.

Proposition III.1. *Given a map* $\mathcal{M} =< \mathcal{X}, \boldsymbol{O}_u, \mathcal{I}, \mathcal{G}, Req >$, *a reference trajectory* ψ, *and a real trajectory* ρ *tracking* ψ *with an error* $\boldsymbol{e} = \rho(t) - \psi(t) \in \mathcal{X}$, ψ *and* ρ *are valid if*

(i) $\psi(0) + \boldsymbol{e} \in \mathcal{I}$
(ii) $\forall t \in [0, D], (\psi(t) + \boldsymbol{e}) \cap \boldsymbol{O}_u = \emptyset$
(iii) $\forall G \in \mathcal{G}, \exists t \in [0, D], (\psi(t) + \boldsymbol{e}) \cap G \neq \emptyset$
(iv) $\forall r \in Req, \psi(t) + \boldsymbol{e} \models r$

Many existing studies synthesize trajectories satisfying Proposition III.1 [22] [23]. Hence, as long as the tracking controller of an autonomous system can keep the tracking errors within the tolerance of the reference trajectories, the real trajectories are guaranteed to be valid too. However, formally verifying whether real trajectories are within the boundary of tracking errors is extremely difficult as the autonomous systems' kinematics is nonlinear, and the environment's reactions are uncertain. In this paper, we combine formal methods, e.g., model checking and testing, in our framework. A correctness-guaranteed mission planner is designed at the high-level layer of the framework, and a learning-based test-case generation is proposed for discovering faults in the real trajectories at the low-level layer.

IV. Framework and Methods

In this section, we introduce our framework for policy synthesis and test-case generation for testing tracking controllers. We also briefly introduce the concrete methods for policy synthesis before we describe our test-case generation methods.

A. The Two-Layer Framework

Figure 4 depicts the two-layer framework that we propose in this paper. The high-level layer is for policy synthesis, where

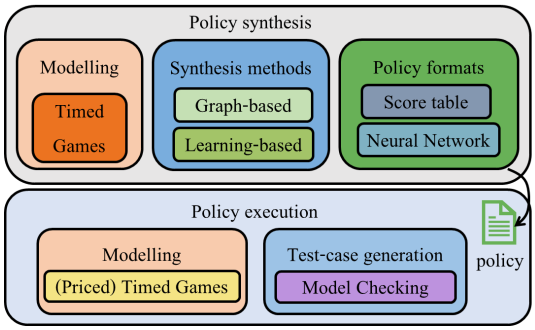


Figure 4. The two-layer framework

the models are timed games, and synthesis methods are based on either graph search [13] or learning [14]. Products of this layer are policies that are either score tables or artificial neural networks. Policies are sent to the low-level layer, where the reference and tracking controllers are modelled as timed games and priced timed games, respectively. Ordinary differential equations (ODE) are used in priced timed games to describe the dynamic function of controllers.

Testing at the low-level layer checks if the tracking errors of the real trajectories ever exceed the tolerance of their reference trajectories and never adjust back within a time frame. Therefore, the test cases we generate at this layer are the sampled points at the reference trajectories, which are periodically fed to the tracking controller.

In some studies [22] [24], a reference trajectory is a single path connecting waypoints. This does not apply to our problem, as executing a policy usually forms multiple reference trajectories due to uncertain environmental reactions. When executing a policy, the reference controller only gets to choose controllable actions and waits for the reaction from the environment, which can be uncertain and result in various trajectories. Therefore, our reference controller has to be aware of the environment's dynamic states (i.e., $state_e$) and generates reference trajectories accordingly, which present a faithful realization of the policy.

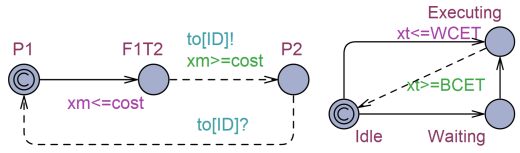
B. Policy Synthesis

We have two ways of synthesizing correctness-guaranteed policies, that is, a graph-search-based method [13] and a

learning-based method [14]. As the focus of this paper is the two-layer framework and the novel method for test-case generation, policy synthesis is presented in Appendix B[1].

1) Models for Policy Synthesis: Models play an essential role in our framework, as they are used in both policy synthesis and test-case generation. Although models for these two aspects are slightly different, they are all timed games and designed in UPPAAL Stratego [8].

We abstract two kinds of actions of the autonomous systems, namely moving and executing tasks, and define two kinds of TG models for policy synthesis, namely movement TG and task-execution TG. However, users of our framework may want to abstract the actions differently, but it would not influence the applicability of our methods. Figure 5 shows



(a) Movement TG (b) Task execution TG

Figure 5. Structures of the TG models.

the structures of the TG models. Locations `P1` and `P2` stand for any valid positions in the environment, and location `F1T2` models the travelling duration (i.e., `cost`) from `P1` to `P2`. Figure 5(b) depicts the task-execution TG that models the best-case-execution time (i.e., `BCET`) and worst-case-execution time (i.e., `WCET`) of the task. In both TG models, controllers only choose when to start the actions of finishing moving and task execution. When to finish, the actions are uncontrollable, as they are performed by the environment. Policy synthesis generates policies that guide autonomous systems to choose controllable actions to eventually reach their goal regardless of how the uncontrollable actions are performed.

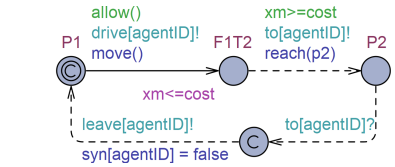
After a policy is synthesized, we test if controllers of the autonomous system can execute the policy correctly, i.e., if its real trajectory ever exceeds the boundary of tracking errors. Next, we introduce the model for the controllers before describing the test-case generation method.

C. Controller Model

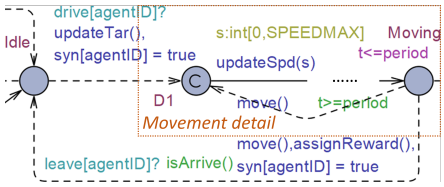
As shown in Figure 3, the tracking controller is responsible for producing commands that control the physical system to follow the reference trajectory. Physical systems are entities in the real world, so our goal is to test tracking controllers where software and hardware can have faults. As a tracking controller periodically takes in points on a reference trajectory as the tracking target, test cases for tracking controllers are reference trajectories. Our test-case generation is model-based, which extends the TG models for policy synthesis. Figure 6 depicts how the movement details are modelled by a TG of reference controllers (RTG)[2]. Figure 6(a) partially shows the adjusted

[1]The appendix will be removed given the paper being accepted.

[2]Complete models are in Appendix A



(a) Adjusted movement TG (MTG)



(b) Reference controller TG (RTG)

Figure 6. Movement models at different levels

TG model for movement (MTG). In `allow()`, MTG searches the policy to see if going from `P1` to `P2` is allowed. When the movement starts, the RTG transfers to a committed location `D1` synchronously with the MTG via channel `drive`. At `D1`, the RTG needs to choose a speed between zero to the maximum speed. This is one movement detail that is not considered in the original MTG because when synthesizing policies travelling times are estimated. Other movement details, such as the turning angle, are modelled similarly, which are not shown in Figure 6(b) for the sake of brevity.

After making a series of decisions, the RTG comes to location `Moving`, from where the model periodically transfers back to location `D1`. The period is set to be a constant number `period`, which means that we sample points on the reference trajectory every *period* time units via function `move()`. After getting back to `D1`, the RTG starts to make a series of decisions again. First, as the travelling time is one of the premises of the policy when the MTG transfers to location `P2`, the RTG also transfers synchronously via channel `leave`. In this way, reference trajectories generated by the RTG hold the policy's premises about travelling time. Second, the function `assignReward()` is called on this edge because the reference controller must ensure that the reference trajectories satisfy several conditions. For example, we may want the system to stop when arriving at `P2`. So, the periodic decision on speed must let the system be able to reach `P2` with `cost` time units and stop (`cost` is defined in MTG). The function `assignReward()` is for assigning rewards when a reference trajectory meets such conditions. In Section IV-D, we introduce how to utilize the rewards to generate valid reference trajectories. RTG for describing the details of task execution is constructed similarly.

As aforementioned, policies can be in various formats, such as score tables and neural networks. When a policy is tabular, we suggest encoding it as a C-code array in Uppaal Stratego. When the system's state space is high-dimensional, such as the Airborne Collision Avoidance System X [25], policies can use large memory space. Neural networks provide a solution to represent large policies in much less memory space [25].

However, neural networks require complex computation, but Uppaal Stratego only supports a subset of the C language. Hence, we suggest compiling the program of a neural network as an external library and calling it in Uppaal Stratego as an external function[3].

In a similar way, PTG for tracking controllers (TPTG) that cope with tracking errors can also be constructed. Though MTG plus RTG are enough for generating test cases, we build TPTG for selecting test cases that are more likely to detect faults. We leave the presentation of TPTG as future work.

D. Model-Based Test-Case Generation

Test-case generation is about automatically creating reference trajectories that hold the four features mentioned in Section III-B. We use the MTG and RTG models in Section IV-C to generate the test cases. The first feature, that is, complying with the physical capability of autonomous systems, is achieved by the constant numbers in the model, which regulate the limits of variables. For example, when choosing a speed, the range of selection is set to be `[0, SPEEDMAX]` (see Figure 6(b). Before describing how the rest of the features are achieved, we introduce the method for test-case generation.

Reference trajectories are supposed to fill the execution details of their policies, such as the movement details in Figure 6. To obtain the execution details, RTG needs to make a series of decisions periodically, such as the speed of movement. However, RTG itself does not know how to make such decisions. Therefore, we assign rewards to the decisions made by RTG and adopt reinforcement learning [7] to synthesize a strategy that guides RTG to make good decisions. Uppaal Stratego provides queries for learning strategies for timed games. Query (1) is one form of these queries.

```
strategy δ = maxE(x)[<=T]{dv}-->{cv}:<> CO
```
(1)

The keyword `maxE(x)` means simulating the model while running the learning algorithm with the purpose of maximizing "`x`", which is a formula for calculating the immediate scores of taking an action at a state. The constant variable `T` is the maximum simulation time, `dv` is a set of expressions regarded as discrete values, and `cv` is a set of expressions regarded as continuous values. The formula "`<> CO`" means only the runs that eventually reach a state holding condition *CO* are selected. As depicted in Figure 7, when running Query (1), Uppaal Stratego simulates the model for a *total* number of runs, among which only a subset of runs are selected (aka. *good* runs) because they reach the states holding condition *CO*. The good runs and their scores (i.e., the value of `x`) are then passed to the learning algorithm, which *partially* observes the states by only looking at the expressions in `dv` and `cv`. The learning algorithm calculates the scores of the state-action pairs in the sampled runs, accumulates the scores, and updates the *strategy*. When a user-defined number of good runs are sampled, or the simulation runs reach the total number, learning finishes and the final *strategy* is considered to be able to guide the model to always eventually satisfy condition `CO`.

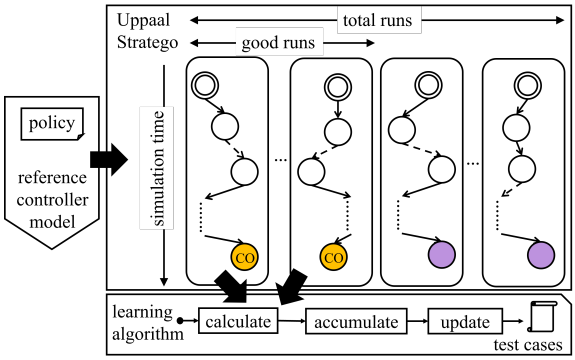


Figure 7. A process of test-case generation. Yellow circles: states holding `CO`. Purple circles: states exceeding the simulation time without holding `CO`.

Now, we set formulate `x` in Query (1) to be the reward that is updated by the function `assignReward()` in the RTG. We set formulate `CO` to be `time` $\geq$ `MAX`, where `time` is a global clock that is never reset, and `MAX` is an estimated time for the system to accomplish its mission. We set the expressions in `dv` to be the discrete variables that represent the sampled points on reference trajectories, and leave `cv` to be empty. Running Query (1) with the model composed of MTG and RTG in Uppaal Stratego gives us a strategy that guides the reference controller on how to make decisions to fill the details of policy execution.

We link an external library to Uppaal Stratego to be the learning algorithm, which receives the input from the model periodically. Besides learning, the external library can also store the outputs of the reference controller, which are in `dv`, for generating test cases. When the learning finishes, we obtain a *decision-making strategy* that contains not only the decisions the reference controllers should make for executing its policy but also the test cases (i.e., outputs of the reference controller) for the tracking controller.

Next, we introduce how the second feature, i.e., holding the premises of policies, is achieved by the synchronization between MTG and RTG. When using queries in the form of Query (1), one must only declare broadcast channels for synchronization, which means that MTG can transit without synchronizing with RTG. Hence, we add an auxiliary variable `syn` for verifying if MTG is always synchronized with RTG when actions start and finish. For example, in Figure 6, when an action of movement starts, the MTG transits from `P1` to `F1T2` while the RTG transits from `Idle` to `D1` too. We name the former transition in MTG the *emitting* transition as it sends the signal of synchronization to RTG and the latter one in RTG the *receiving* transition. In Uppaal Stratego, synchronous transitions take place consecutively with the values of clocks unchanged, and updates on emitting transitions always happen before the ones on receiving transitions. So, when we flip the auxiliary variable `syn` to *false* at emitting transitions in MTG

[3]This feature is currently only supported in the Stratego version of Uppaal, version 4.1.20-7 or later. See *https://docs.uppaal.org/*.

and then flip its value back to *true* at the receiving transitions in RTG, the value of `syn` should always be *true* if MTG and RTG are always synchronized on the actions of starting and finishing actions. Formally, we verify the following query, in which `ValidID` represents integers ranging from the ID of the first autonomous system to the ID of the last one.

$$\texttt{A[] forall (id:ValidID) syn[id] under } \delta \quad (2)$$

The keyword `under` forces the model checker to call the external library where the decision-making strategy δ is stored. Specifically, when the model checker encounters multiple controllable actions, it calls the external library, which returns the actions with the best score. Next, instead of exploring all the actions in classic model checking, the model checker only chooses the actions returned from the external library. In this way, the model's behaviour is *controlled* by the external library. If Query (2) is satisfied, we can be sure that MTG and RTG are always synchronized when they are under the control of strategy δ. Hence, the premises of policies, such as the time intervals of performing actions, are held by RTG.

The third feature, that is, realization and faithfulness of the reference trajectories, is achieved for the following reasons.

1) Realization: Realization requires that for every path in a policy, there is a reference trajectory executing the path. The difficulty in achieving this feature is that if the reference controller's policy and decision-making strategy are synthesized by learning, the results may contain unuseful information as the learning is based on a random simulation. In our previous work, we propose a method for compressing strategies in Uppaal Stratego by removing unuseful state-action pairs [14]. We adopt that method to guarantee that the policies and decision-making strategies of reference controllers only contain the state-action pairs that accomplish the mission eventually. Assume $\mathcal{C}$ is composed of MTG and $\mathcal{C}^*$ is composed of MTG and RTG, both $\mathcal{C}$ and $\mathcal{C}^*$ have a Boolean variable m that turns *true* only when the mission is accomplished, policy σ is synthesized by $\mathcal{C}$ and δ is the decision-making strategy synthesized by $\mathcal{C}^*$, the following queries can remove the unuseful state-action pairs in σ and δ, where $\mathcal{C}^*|\sigma$ means σ is encoded in $\mathcal{C}^*$ (e.g., the `allow` function in Figure 6(a)).

$$\texttt{A<> } \mathcal{C}.m \texttt{ under } \sigma \quad (3)$$

$$\texttt{A<> } \mathcal{C}^*|\sigma.m \texttt{ under } \delta \quad (4)$$

Similar to the verification of Query (2), when verifying Queries (3) and (4), the model checker explores the model state space exhaustively by consulting the external library until reaching a loop, a state that has no succeeding states, or a state where $\mathcal{C}.m$ or $\mathcal{C}^*|\sigma.m$ is *true*, respectively. In the last case, the query is satisfied, which means that the model can always eventually accomplishes the mission regardless of how the environment behaves. Additionally, in the external library, we mark the visited state-action pairs. So, once the query is satisfied, the marked pairs are sufficient to guide the model toward mission accomplishment. Last, the unmarked pairs are considered as *unuseful* information that is removed from the strategy. In essence, if Query (3) is satisfied, all and only the paths that eventually accomplish the mission are included in σ. If Query (4) is satisfied, all and only the decisions under the control of σ and δ are chosen by the reference controller, and they also guarantee to eventually accomplish the mission. Collectively, if Query (2) to Query (4) are satisfied, δ stores the sampled points of the reference trajectories that realize σ.

2) Faithfulness: Faithfulness requires that every reference trajectory executes a corresponding path in the policy. Similar to the reasoning of realization, faithfulness is also guaranteed by the satisfaction of Queries (2) to (4).

The last feature of reference trajectories is the satisfaction of Proposition III.1. As aforementioned, there are many studies about synthesizing trajectories satisfying the proposition. Hence, in this paper, we utilize the concept of *safe envelope* that is proposed in literature [23]. Essentially, a safe envelope of a reference trajectory is a region surrounding the trajectory as an axe. As long as the dynamic function of tracking errors has a *Lyapunov function*, they are bounded within the safe envelope. We use the width of the safe envelope as the tolerance of our reference trajectories (i.e., $\boldsymbol{e}$ in Proposition III.1). The test cases here are for testing whether a tracking controller goes outside the safe envelope of its reference trajectory.

V. A Case Study on An Autonomous Quarry

The methods for policy synthesis are evaluated in our previous studies in an industrial use case of an autonomous quarry [13] [14]. In this section, we evaluate the performance of the test-case generation method in this use case provided by Volvo Construction Equipment, Sweden.

A. Case Description

Trucks and wheel loaders carry out their missions autonomously in an autonomous quarry. Wheel loaders are responsible for digging stones at stone piles and loading them into trucks. The latter carries the stones to a primary crusher and unloads them. After that, trucks transport the crashed stones to a secondary crusher and unload them again. An autonomous quarry is supposed to work 24 hours per day without human intervention. Hence, they need to plan their paths and schedule their tasks so that all stones are eventually carried to the secondary crusher. In this case, we re-use a scenario created in our previous experiments [14], where we showed how the policy synthesis is accomplished. In this scenario, we have one wheel loader, one truck, two primary crushers that the trucks need to choose from, and one secondary crusher. A policy is synthesized, and in the next subsection, we introduce how test cases are generated.

B. Test-Case Generation

The synthesized policy regulates the truck to move to a stone pile first, collaborate with the wheel loader there to get unloaded with stones, and then move to a primary crusher to unload stones before it finishes the mission at the secondary crusher. Now, the reference trajectories of this policy must

fill in the details of movement so that the truck knows when to accelerate or brake so that it can reach each of the task positions at a proper speed (e.g., fully stopped) and the direction of turning when it needs to avoid obstacles.

We design a model based on the template in Figure 6, train it in Uppaal Stratego using Q learning and obtain a strategy that guides the truck to adapt its speed and moving direction. Next, we remove the unuseful information in the strategy and store the strategy as a JSON (JavaScript Object Notation) file. Then, we parse this JSON file in a Java program, where the kinematics and tracking controller of the truck is also programmed. Therefore, the JSON file stores the test cases, and the Java program is the system under test in this case study. The ordinary differential equations of the truck's kinematics and tracking errors are presented in Appendix C.

C. Experimental Results and Discussion

We evaluate two aspects of the test-case generation method, i.e., *efficiency* and *effectiveness*. *Efficiency* refers to the generation time per ten test cases, and *effectiveness* refers to the number of faults detected in the same system under test. As we are looking for the test cases (i.e., reference trajectories) that faithfully realize the execution of a policy, the results must satisfy properties in the forms of Queries (2) - (4). Hence, generation time refers to synthesizing a strategy that satisfies the properties but does not include the verification time. However, the verification is considerably short (i.e., within a second) for all the cases. Additionally, the periods of sampling points on reference trajectories (i.e., `period` in Figure 6(b)) reflect the granularity of calling the learning algorithm. Short periods mean accurate sampling on the reference trajectories and, thus, a long time for learning.

Table I
EXPERIMENTAL RESULTS SHOWING THE EFFICIENCY AND EFFECTIVENESS OF THE TEST-CASE GENERATION METHOD

period	efficiency	effectiveness
10	1.5 s/10 cases	1 fault
5	4.2 s/10 cases	2 faults
1	26.4 s/10 cases	11 faults

Table I presents the experimental results. Efficiency drops 17.6 times (i.e., 26.4/1.5) when the length of periods decreases ten times (i.e., 10/1), showing that learning becomes much more difficult when the sampling periods are shorter. However, the effort of learning is not in vain, as effectiveness increases when the sampling period becomes short. The faults are the points where tracking errors exceed the threshold, and the tracking controller cannot get the system back into the boundary of track errors within a time frame.

In summary, our test cases are guaranteed to cover the entire policy as the generated reference trajectories satisfy properties in the forms of Queries (2) - (4). The generation time of test cases is short considering the difficulty of the job, i.e., learning comprehensive strategies that faithfully realize the execution of a policy. The test cases can detect faults in the system under test. Via an observation of the faults, we conclude that when the turning angle changes dramatically, the tracking controller tends to lose control over the system, failing to restrict the tracking errors within a threshold.

VI. Related Work

Testing for multi-agent systems (MAS) is a widely studied area. Araujo et al. [26] conducted a systematic review of autonomous systems testing, presenting the state-of-the-art solutions and research gaps in this area. Our framework fills the gap of a holistic framework that combines the discrete and continuous aspects of the systems. The formal qualitative analysis of our generated test cases is another contribution that fills a gap. Tao et al. [27] designed a method for generating test cases based on ontology. They use UML as the modelling language, whereas we use formal models that guarantee the quality of the generated test cases. Gonçalves et al. [28] proposed a framework named *CPN4M* for testing MAS. *CPN4M* employs coloured Petri net and generates test cases for MAS social level. This work is orthogonal to our study as they aim to find organizational errors, such as when agents lack resources to achieve goals. In contrast, our study focuses on the controlling aspect of MAS.

Regarding the frameworks and tools for autonomous systems, D'Urso et al. [29] designed an integrated framework to simulate multiple autonomous systems. Their framework provides a versatile tool for simulation and high-level logic implementing strategies and control of autonomous systems. Our framework provides a separation of concerns by splitting the problem into two layers, supported by different formal modelling and verification methods. Bersani et al. [30] present the tool PuRSUE (Planner for RobotS in Uncontrollable Environments), which supports synthesizing high-level runtime control strategies for robotic applications. Their tool is based on Uppaal Tiga, which uses graph-search-based synthesis. Compared to these works, our study provides a holistic framework for policy synthesis and testing, which has not been seen in this research field.

VII. Conclusions and Future Work

We propose a two-layer framework for policy synthesis and test-case generation. The framework separates the discrete policy synthesis and policy continuous execution into two layers, supported by different and suitable formal methods. The policy synthesis is solved by two kinds of methods: graph-search-based and learning-based methods. The results of both kinds of synthesis have a correctness guarantee. For testing the policy execution, we design a novel approach for test-case generation. The method employs reinforcement learning and model checking to find all the reference trajectories that faithfully realize the execution of a policy.

The next step of this study is to model tracking controllers in our framework for selecting error-prone test cases. Experimenting with other learning algorithms for test-case generation is also an interesting direction of research. Constructing the framework upon an existing platform for MAS, e.g., GAMA [31], is another future work direction.

REFERENCES

[1] S. Rabin, "Game programming gems, chapter a* aesthetic optimizations," *Charles River Media*, 2000.

[2] M. Fisher, V. Mascardi, K. Y. Rozier, B.-H. Schlingloff, M. Winikoff, and N. Yorke-Smith, "Towards a framework for certification of reliable autonomous systems," *Autonomous Agents and Multi-Agent Systems*, vol. 35, no. 1, pp. 1–65, 2021.

[3] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2008.

[4] M. Shaw and D. Garlan, *Software architecture: perspectives on an emerging discipline*. Prentice Hall Englewood Cliffs, 1996, vol. 1.

[5] R. Gu, R. Marinescu, C. Seceleanu, and K. Lundqvist, "Towards a two-layer framework for verifying autonomous vehicles," in *NASA Formal Methods Symposium*. Springer, 2019, pp. 186–203.

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

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

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

[9] M. Hendriks, W. Yi, P. Petterson, J. Hakansson, K. Larsen, A. David, and G. Behrmann, "Uppaal 4.0," in *Third International Conference on the Quantitative Evaluation of Systems - (QEST'06)*, 2006.

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

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

[12] M. Jaeger, P. G. Jensen, K. Guldstrand Larsen, A. Legay, S. Sedwards, and 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.

[13] R. Gu, P. G. Jensen, D. B. Poulsen, C. Seceleanu, E. Enoiu, and K. Lundqvist, "Verifiable strategy synthesis for multiple autonomous agents: a scalable approach," *International Journal on Software Tools for Technology Transfer*, vol. 24, no. 3, pp. 395–414, 2022.

[14] R. Gu, P. G. Jensen, C. Seceleanu, E. Enoiu, and K. Lundqvist, "Correctness-guaranteed strategy synthesis and compression for multi-agent autonomous systems," *Science of Computer Programming*, vol. 224, p. 102894, 2022.

[15] R. Alur and D. Dill, "Automata for Modeling Real-time Systems," in *Automata, languages and programming*. Springer, 1990, pp. 322–335.

[16] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime, "Uppaal tiga user-manual," *Aalborg University*, 2007.

[17] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[18] A. Arcuri, "An experience report on applying software testing academic results in industry: we need usable automated test generation," *Empirical Software Engineering*, vol. 23, no. 4, pp. 1959–1981, 2018.

[19] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*. Elsevier, 2010.

[20] B. I. Kazem, A. I. Mahdi, and A. T. Oudah, "Motion planning for a robot arm by using genetic algorithm," *Jjmie*, vol. 2, no. 3, pp. 131–136, 2008.

[21] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[22] C. Fan, Z. Qin, U. Mathur, Q. Ning, S. Mitra, and M. Viswanathan, "Controller synthesis for linear system with reach-avoid specifications," *IEEE Transactions on Automatic Control*, 2021.

[23] D. Sun, J. Chen, S. Mitra, and C. Fan, "Multi-agent motion planning from signal temporal logic specifications," *IEEE Robotics and Automation Letters*, vol. 7, no. 2, pp. 3451–3458, 2022.

[24] S. L. Herbert, M. Chen, S. Han, S. Bansal, J. F. Fisac, and C. J. Tomlin, "Fastrack: A modular framework for fast and guaranteed safe motion planning," in *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*. IEEE, 2017, pp. 1517–1522.

[25] K. D. Julian, M. J. Kochenderfer, and M. P. Owen, "Deep neural network compression for aircraft collision avoidance systems," *Journal of Guidance, Control, and Dynamics*, vol. 42, no. 3, pp. 598–608, 2019.

[26] H. Araujo, M. R. Mousavi, and M. Varshosaz, "Testing, validation, and verification of robotic and autonomous systems: A systematic review," *ACM Transactions on Software Engineering and Methodology*, 2022.

[27] J. Tao, Y. Li, F. Wotawa, H. Felbinger, and M. Nica, "On the industrial application of combinatorial testing for autonomous driving functions," in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2019, pp. 234–240.

[28] E. M. N. Gonçalves, R. A. Machado, B. C. Rodrigues, and D. Adamatti, "Cpn4m: Testing multi-agent systems under organizational model m oise+ using colored petri nets," *Applied Sciences*, vol. 12, no. 12, p. 5857, 2022.

[29] F. D'Urso, C. Santoro, and F. F. Santoro, "An integrated framework for the realistic simulation of multi-uav applications," *Computers & Electrical Engineering*, vol. 74, pp. 196–209, 2019.

[30] M. M. Bersani, M. Soldo, C. Menghi, P. Pelliccione, and M. Rossi, "Pursue-from specification of robotic environments to synthesis of controllers," *Formal Aspects of Computing*, vol. 32, no. 2, pp. 187–227, 2020.

[31] Gama platform. http://gama-platform.org.

[32] R. Gu, E. Enoiu, and C. Seceleanu, "Tamaa: Uppaal-based mission planning for autonomous agents," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020, pp. 1624–1633.

[33] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Progress on the state explosion problem in model checking," in *Informatics*. Springer, 2001, pp. 176–194.

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

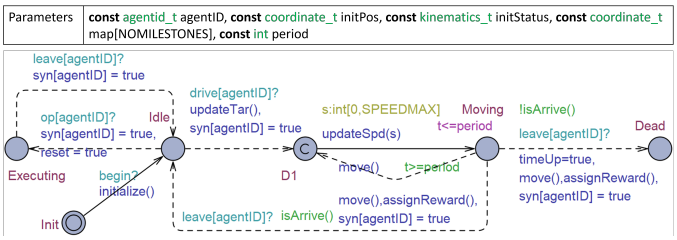


Figure 11. A reference controller TG that fills in the movement details (this is the complete model of RTG in Figure 6(b))

APPENDIX

A. Complete Model Templates

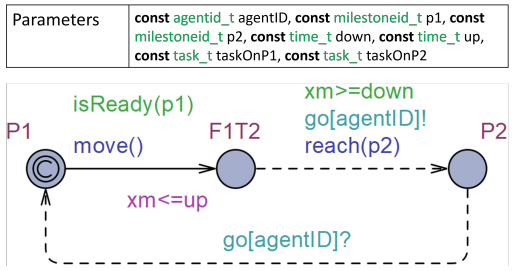


Figure 8. A movement TG for policy synthesis (this is the complete model of movement TG in Figure 5(a))

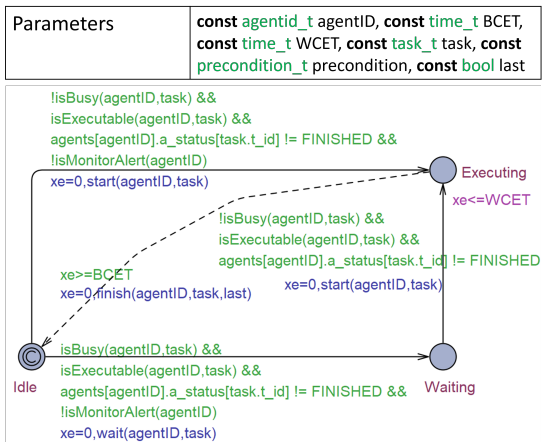


Figure 9. A task execution TG for policy synthesis (this is the complete model of task-execution TG in Figure 5(b))

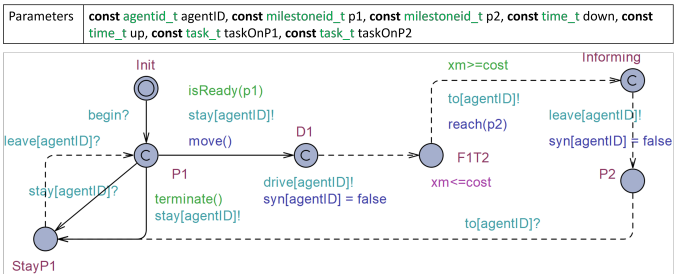


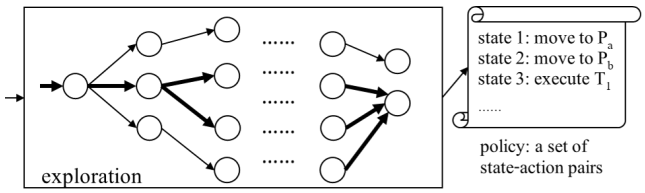
Figure 10. A movement TG for test-case generation (this is the complete model of MTG in Figure 6(a))

B. Policy Synthesis

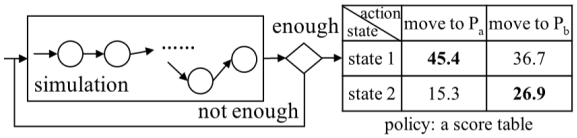
There are two ways of policy synthesis in our framework, namely graph-search-based synthesis [32] and learning-based synthesis [13]. As depicted in Figure 12(a), the graph-search-based synthesis explores the state space of TG exhaustively and finds the traces that accomplish the mission eventually while satisfying other requirements. As the state-space exploration here is exhaustive, the graph-search-based synthesis can guarantee the results to be correct but it suffers from the notorious state-space-explosion problem [33]. When the number of models becomes large, the method fails to generate a result in a reasonable time [32]. Hence, we propose an alternative method based on reinforcement learning [7].

As depicted in Figure 12(b), learning-based synthesis uses random simulation to explore the state-space of a system model instead of an exhaustive state-space exploration. Traces are sampled from the simulation and fed into a learning algorithm, such as Q learning [34], together with scores of the traces. Gradually, the learning algorithm produces a score table of state-action pairs where actions with the highest score at each of the states are considered as the *optimal* solution that would guide the systems to accomplish their missions.

Regardless of which method for policy synthesis is chosen, the actual execution of the policies can go wrong because of the inevitable uncertainties existing in the systems and environment, such as tracking errors. Unfortunately, formally verifying the real trajectories of autonomous systems is an undecidable problem. Therefore, we turn to test generation and execution which provide a means of detecting possible errors in the actual execution of policies. Among all the testing activities, test-case generation usually costs the most time and effort. In the next subsections, we introduce our controller models and how to generate test cases automatically.



(a) Graph-search-based synthesis



(b) Learning-based synthesis

Figure 12. Methods for policy synthesis.

C. Ordinary Differential Equations in Case Study

The autonomous quarry in our case study is a 2D map, where the kinematics of the trucks are given by the following equation, where x and y are the coordinates, θ is the moving direction, v is the velocity, and w is the angular velocity.

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} cos(\theta) & 0 \\ sin(\theta) & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ w \end{bmatrix} \tag{5}$$

Tracking errors are the differences between the reference trajectory and the real trajectory in three dimensions, that is, x, y, and θ. The calculation of tracking errors is straightforward and given by the following equation.

$$\begin{bmatrix} e_x \\ e_y \\ e_\theta \end{bmatrix} = \begin{bmatrix} cos(\theta) & sin(\theta) & 0 \\ -sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{ref} - x \\ y_{ref} - y \\ \theta_{ref} - \theta \end{bmatrix} \tag{6}$$

The error dynamics can be various and we choose the following equation to describe it.

$$\begin{bmatrix} \dot{e_x} \\ \dot{e_y} \\ \dot{e_\theta} \end{bmatrix} = \begin{bmatrix} we_y - v + v_{ref} cos(e_\theta) \\ -we_x + v_{ref} sin(e_\theta) \\ w_{ref} - w \end{bmatrix} \tag{7}$$

The tracking controller produces the value of v and w to compensate for the tracking errors of x, y, and θ. The following control law is chosen.

$$v = v_{ref} cos(e_\theta) + e_x \tag{8}$$

$$w = w_{ref} + v_{ref}(e_y + sin(e_\theta)) \tag{9}$$

According to the literature [22], the tracking errors have a Lyapunov function that is given by the following equation.

$$V = 1 + \frac{1}{2}(e_x^2 + e_y^2) - cos(e_\theta) \tag{10}$$

Therefore, the tracking error of each of the line segments of the reference trajectory is upper bounded $\sqrt{(l^2 + 4i)}$, where l is the initial deviation from the reference trajectory and i is the index of the line segments. We decrease this value in our Java program to set a more restrictive boundary and test if the real trajectory ever deviates from the reference trajectory further than that boundary.