Layered controller synthesis for dynamic multi-agent systems*

Emily Clement[1], Nicolas Perrin-Gilbert[1], and Philipp Schlehuber[2]

[1] Sorbonne Université, CNRS, Institut des Systèmes Intelligents et de Robotique, ISIR, F-75005 Paris, France `lastname@sorbonne-universite.fr`
[2] EPITA Research Laboratory `firstname@lrde.epita.fr`

Abstract. In this paper we present a layered approach for multi-agent control problem, decomposed into three stages, each building upon the results of the previous one. First, a high-level plan for a coarse abstraction of the system is computed, relying on a parametric timed automata augmented with stopwatches as they allow to efficiently model simplified dynamics of such systems. In the second stage, the high-level plan, based on SMT-formulation, mainly handles the combinatorial aspects of the problem, provides a more dynamically accurate solution. These stages are collectively referred to as the SWA-SMT solver. They are correct by construction but lack a crucial feature: they cannot be executed in real time. To overcome this, we use SWA-SMT solutions as the initial training dataset for our last stage, which aims at obtaining a neural network control policy. We use reinforcement learning to train the policy, and show that the initial dataset is crucial for the overall success of the method.

1 Introduction

Controlling a system involving multiple agents sharing a common task is a problem occurring in several domains such as mobile or industrial robotics. Concrete instances range from controlling swarms of drones, autonomous vehicles or warehouse robots. The problem is studied for specific instances but remains a difficult problem in general, especially in safety critical cases. The main complexity stems from the different types of decisions to take: such control problems often have a strong combinatorial side while the approach also has to deal with the physical reality of the agents, whose state typically evolves according to some differential equation, and limitations on the control inputs have to be taken into account. Finally, the controller has to be executed in real-time, which typically limits the applicability of formal methods due to their high complexity.

In this paper, we propose a layered approach for synthesizing control strategies for multi-agent dynamical systems, whose effectiveness we demonstrate on an example of centralized traffic guidance used for illustration throughout the paper.

*This work was partially funded by ANR project TickTac (ANR-18-CE40-0015).

The layered approach involves three stages, each one addressing a specific aspect of the control problem by building on the results of the previous stage. The first stage deals with the combinatorial side of the control problem: using a sufficiently coarse abstraction of the system dynamics, one can rely on timed automata augmented with stopwatches as a model. Efficient algorithms exist to explore such models allowing us to find a high-level plan that guarantees success in this abstract setting. The second stage takes the high-level plan as an input and refines it using a more realistic model of the system while maintaining a high degree of similarity between the refined and high-level solution. In our running example, we formulate this as an SMT problem, respecting the discrete version of the differential equation describing the system while also taking into account the input and state constraints. The complexity of this stage remains reasonable as the combinatorial aspects have already been solved. The final stage addresses the issue of real-time execution and generalization. To this end, we train a neural network policy via reinforcement learning. We use the two first stages to construct a dataset of successful episodes on many random instances of the problem, and exploit this dataset to guide the reinforcement learning towards good solutions. On our running example, we show that the initial dataset of solutions is crucial for the overall success: the reinforcement learning only succeeds if it has access to it.

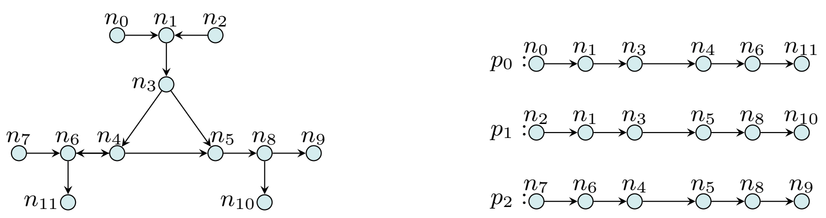
The rest of the paper is structured as follows. In section 2 we present our running example, section 3 briefly discusses related work, and then we describe each stage of the approach in its own section along with the necessary technical background: the first stage using timed automata in section 4, the second SMT-based stage in section 5 and finally the synthesis of the actual controller based on reinforcement learning in section 6.

2 Running example: centralized traffic control

Let us first present a multi-agent system used as running example (fig 1a) to illustrate our method throughout the article. In this example, each agent models a physical car on a road network. Each of the cars is given a fixed path to follow and it needs to attain its designated goal position from its initial position, while maintaining a security distance to the other cars. In such a setting the dynamics can be reduced to a second order ordinary differential equation with lower and upper bounds on the first and second derivative (corresponding to the speed and acceleration of the car).

2.1 Multi-agent traffic

We model the traffic as a network of sections on which a fixed number of cars have to navigate. Cars drive along their *paths* which consist of a list of sections. Cars cannot overtake or cross each other on the same section. A minimum security distance must be maintained at all time.


n_0
n_1
n_2
n_3
n_7
n_6
n_4
n_5
n_8
n_9
n_{11}
n_{10}
p_0 : n_0 n_1 n_3 n_4 n_6 n_{11}
p_1 : n_2 n_1 n_3 n_5 n_8 n_{10}
p_2 : n_7 n_6 n_4 n_5 n_8 n_9

(a) A 2-D representation of the traffic. (b) Three different paths p_0, p_1, p_2.

Fig. 1: Our running example, arrows indicate which direction can be taken.

To define a **section**, denoted s, we specify its beginning node, n_b, its end node, n_e, and its length L. We can therefore write a section $s := s_{[n_b,n_e],L}$. To specify its direction ($\rightarrowtriangle$ or $\leftarrowtriangle$), we augment a section s with a direction $\boldsymbol{d} \in \{\rightarrowtriangle, \leftarrowtriangle\}$ into a **directed section**, denoted $\boldsymbol{s}$, or $(s, \boldsymbol{d})$. The notion of beginning and end nodes of a section is extended to directed sections, reversing the two nodes if $\boldsymbol{d} = \leftarrowtriangle$. A directed section $\boldsymbol{s'}$ is a **successor** (*resp.* **predecessor**) of the directed section $\boldsymbol{s}$ if $n_e = n'_b$ (*resp.* $n'_e = n_b$). The sections $\boldsymbol{s}$ and $\boldsymbol{s'}$ are said to be **neighbours** if $\boldsymbol{s}$ is either a **successor** or **predecessor** of $\boldsymbol{s'}$.

A **path** p is defined as a finite list of directed sections: $p = (\boldsymbol{s_k})_{0 \le k \le m}$ such that for all $k \in [0, m-1]$, $\boldsymbol{s_{k+1}}$ is a successor of $\boldsymbol{s_k}$. The end (*resp.* beginning) node of a path is the end (*resp.* beginning) node of its last (*resp.* first) directed section. By abuse of notation, we denote $\boldsymbol{s} \in p$ if there exists an index k such that $\boldsymbol{s} = \boldsymbol{s_k}$.

In fig 1b, 3 paths are described: directed sections $(s_{[n_4,n_6],L}, \leftarrowtriangle)$ and $(s_{[n_4,n_5],L}, \rightarrowtriangle)$ are neighbours, but $(s_{[n_4,n_6],L}, \rightarrowtriangle)$ and $(s_{[n_4,n_5],L}, \rightarrowtriangle)$ are not.

A **car** is defined as a tuple of an index i and a path p. A **car traffic**, denoted $\mathcal{C}$, is a set of cars. The set of sections (*resp.* directed sections) of the cars of a car traffic is denoted $\mathcal{S}$ (*resp.* $\boldsymbol{\mathcal{S}}$). A section s is an **intersection** if there exist two different paths p, p' and two directions $\boldsymbol{d}, \boldsymbol{d'} \in \{\rightarrowtriangle, \leftarrowtriangle\}$ such that $(s, \boldsymbol{d}) \in p, (s, \boldsymbol{d'}) \in p'$.

Let us illustrate again with our example of fig 1 with a car traffic composed of three cars $(i, p_i)_{i=0,1,2}$. Intersections here are $s_{[n_1,n_3],L}$, $s_{[n_4,n_6],L}$ and $s_{[n_5,n_8],L}$.

Since the path of each car is fixed, we only need to keep track of its speed and its progress along its path.

2.2 Collision avoidance problem

Given a security distance ε, the initial positions of all cars, bounds on their speed, acceleration and deceleration, the goal is to find trajectories for all cars, which can be interpreted as a **centralized strategy**, that respect the three following rules.

Let $\text{sect}(c,t)$ (*resp.* $\text{sect}_d(c,t)$) denote the current section (*resp.* directed section) of car c at time t and $p_{\boldsymbol{s}}(c,t)$ its current relative position within $\boldsymbol{s}$.

1. **Same directed section**: for all cars $c_i, c_j \in \mathcal{C}$, $c_i \neq c_j$, for all $t \geq 0$, if $\text{sect}_d(c,t) = \text{sect}_d(c',t) = \boldsymbol{s}$ then: $|p_{\boldsymbol{s}}(c_i,t) - p_{\boldsymbol{s}}(c_j,t)| \geq \varepsilon$

2. **Neighbouring sections**: for all cars $c_i = (i, [\cdots, (s', \boldsymbol{d}'_i), \cdots]) \in \mathcal{C}$, if there exists a car $c_j = (j, [\cdots, (s, \boldsymbol{d}_j), (s', \boldsymbol{d}'_j), (s'', \boldsymbol{d}''_j), \cdots]) \in \mathcal{C}$ we have two cases:

– $\boldsymbol{d}'_i = \boldsymbol{d}'_j$: then for all t s.t. $\text{sect}_d(c_i,t) = (s', \boldsymbol{d}'_i)$, $\text{sect}_d(c_j,t) = (s'', \boldsymbol{d}''_j)$ we have $L' - p_{(s',\boldsymbol{d}'_i)}(c_i,t) + p_{(s'',\boldsymbol{d}''_j)}(c_j,t) \geq \varepsilon$.

– $\boldsymbol{d}'_i \neq \boldsymbol{d}'_j$: then for all t s.t. $\text{sect}_d(c_i,t) = (s', \boldsymbol{d}'_i)$, $\text{sect}_d(c_j,t) = (s, \boldsymbol{d}_j)$ we have $L' - p_{(s',\boldsymbol{d}'_i)}(c_i,t) + L - p_{(s,\boldsymbol{d}_i)}(c_j,t) \geq \varepsilon$

3. **Same section, opposite direction**: for all section $s \in \mathcal{S}$, for all $t \geq 0$ and for each pair of cars $c_i, c_j \in \mathcal{C}$: $\neg(\text{sect}_d(c_i,t) = (s, \rightarrow) \wedge \text{sect}_d(c_j,t) = (s, \leftarrow))$

2.3 Running example

Let us consider three possible paths, illustrated in fig 1. All sections have the same security distance ε and the same length, $L = 30$, except for those from n_3 to n_5 and n_3 to n_4 that have length $30\sqrt{2}$. The cars are defined as $(i, p_0)_{i=1,2,3}$, $(i, p_1)_{i=4,5,6}$ and $(i, p_2)_{i=7,8,9}$. They all have different initial and goal positions, starting with a security distance 2ε between them as shown in fig 2. For instance, the initial position of car 2 is 2ε to the right of n_0 in direction of n_1. Its goal position is $L - 2\varepsilon$ to the right of n_6. All other cars are setup similarly. Let us precise that, to make the implementation of the car traffic more convenient, we created additional nodes dedicated to the initial and goal positions of each car, omitted here for clarity. Therefore in our actual implementation, the section from n_0 to n_1 is subdivided with nodes n'_0 and n''_0 representing the actual initial positions of car 2 and 3.

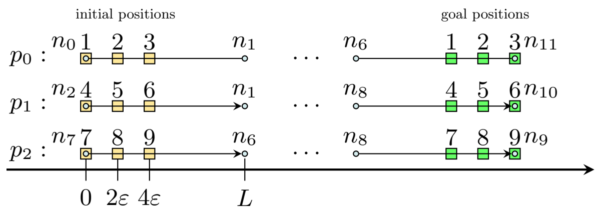


Fig. 2: Initial and goal position of cars

In the next sections, we present our method and apply it to this problem with several levels of abstraction. Firstly, in section 4, we conceive a high-level plan relying on timed automata, assuming that the speed of cars is discretized to two values, a nominal value v and 0, and that cars can switch between these speeds instantly. We also assume that all vehicles must respect the same the security distance on all sections (extending to different speed or security distance between sections is trivial). Secondly, in section 5, we refine this abstract model by allowing for arbitrarily many speed levels between v and 0 and also by respecting the maximal values for acceleration and deceleration. In section 6, there are discrete time steps, but the neural network policy outputs continuous values for the acceleration/deceleration.

3 Related work

There exists a rich literature on multi-agent systems, including path planning or controller synthesis. Providing a general overview over these topics is beyond the scope of this paper, however we give a brief overview of the ones matching our objective the best. In our model, decisions are taken by a centralized control agent (the reference trajectories) with perfect knowledge and then executed by all executing agents (the cars). For a good survey of topics related to multi-agent systems we refer to [15].

Path planning, collision avoidance and controller synthesis for multi-agent systems in general as well as centralized traffic control allow for a rich variety of useful abstraction which in turn leads to a large spectrum of techniques and concepts that can be applied. Approaches can be differentiated into different categories: fully discretized approaches largely ignoring the underlying dynamics of the system fall into the category of multi-agent path finding. Here the problem boils down to a graph search on (very) large graphs, see [32, 31]. Multi-agent motion planning in contrast takes into account the underlying the dynamics (possibly even uncertainty) and works over continuous domains as done in [13]. We are positioned in between these approaches: we consider the dynamics of the system, however the road system is fixed, so the planning is over a finite domain.

Another way to distinguish approaches is the complexity of the specification. If the target is known and the only goal is to avoid collision, less formal approaches ensuring safety and success in practice without a proof can provide good results as shown in [16, 10]. More complex specifications are taken into account by works like [26] and [11]. In [26], controllers for drones verifying temporal logic specifications are synthesized given low-level controllers guaranteeing to bring them from one region to another exist; collision between different drones is ignored as they are supposed to fly at different altitudes. In [11] controllers for a fleet of warehouse robots that have to share resources to fulfill different task in a near optimal manner are learned. Our final layer shares some characteristics of [10]: it does not provide formal guarantees but is executable in real-time. With [11] we share the idea of a layered approach however to achieve different goals. In their work one controller is learned for resource distribution and another for path-planning. Our approach in contrast uses layers to refine plans with different levels of abstractions. In [27], temporal logic task specifications are translated into real-valued functions that can be used as reward signals to guide reinforcement learning.

Finally, there exist also many works tackling explicitly collision avoidance in traffic scenarios like [23, 21, 22, 14, 28]. The works of [23, 21, 22] rely on discretization and overapproximation, then proving collision freeness for a given set controllers and maneuvers in an offline manner. In [14] only intersections are treated and not the problem of sharing a section while driving in the same direction. [28] avoids collision by finding an optimal scheduling for the traffic lights, which is however only applicable in a very restricted scenario.

4 High level planning

Let us present the first level of our method, based on Timed Automata (TA), a well established tool to model real-time systems with timing constraints using clocks. Besides time, these clocks can also be used to model other quantities if they behave somewhat similarly to time in an abstract sense. In our running example, we assign to each car a clock that tracks its progress along its path. This basic framework is not expressive enough to obtain a useful model and we need to use several extensions. We augment the TA with stopwatches to represent the agent at standstill or driving at nominal speed. We rely on channels to ensure the collision avoidance and relative order between cars. To obtain reachability in minimal time, we dedicate a clock to represent global time (therefore never stopped nor reset), which is used as a parameter that does not appear in the constraints.

4.1 Timed Automata, stopwatches and channels

In this section we recall standard TA semantics as well as the needed extensions.

Definition 1 ([3]). *A Timed Automaton (TA) $\mathcal{A}$ can be defined by the tuple $(Q, \ell_0, \mathcal{X}, Inv, \Sigma, T)$, where Q is a finite set of locations, ℓ_0 is the initial location, $\mathcal{X}$ is a finite set of clocks, Σ a finite alphabet, $Inv : Q \to \mathcal{G}(\mathcal{X})$ is the function of invariants of locations and $T \subseteq Q \times \mathcal{G}(\mathcal{X}) \times \Sigma \times \mathcal{R}(\mathcal{X}) \times Q$ a set of transitions.*

There are two types of transitions with the following semantics.

- *Time elapsing move.* $(\ell, v) \xrightarrow{\delta} (\ell, v')$*: elapses some amount of time δ by setting $v' = v + \delta$ and is only allowed if $v \models Inv(\ell)$ and $v' \models Inv(\ell)$*
- *Discrete transition.* $(\ell, v) \to (\ell', v')$ *indicates a discrete transition. This is only possible if **(1)** $v \models Inv(\ell)$ and $v' \models Inv(\ell')$ and **(2)** there exists a transition $t := (v, g, a, r, \ell') \in T$ such that $v \models Inv(\ell)$, $v \models g$, $v' = v\,[r \leftarrow 0]$ and $v' \models Inv(\ell')$. We say that t is labelled by a.*

Here, $v + \delta$ denotes the valuation v_δ such that for any clock x in the set of clocks, denoted $\mathcal{X}$, $v_\delta(x) = v(x) + \delta$ and $v\,[r \leftarrow 0]$ the valuation v_r such that for

$v_r(x) = 0$ if $x \in r$ and $v(x)$ otherwise. $\mathcal{G}(\mathcal{X})$ defines the set of clock-constraints by a conjunction of simple (in-) equalities: $\wedge_i x \bowtie c_i$ for some clock $x \in \mathcal{X}$, some constant $c_i \in \mathbb{N}$ and $\bowtie \in \{\leq, <, =, >, \geq\}$. We denote $v \models g$ to express that a valuation v satisfies the constraint of $g \in \mathcal{G}(\mathcal{X})$.

Location based Stopwatch Timed Automata (SWA), presented in [2], are an extension of TA where clocks can be "frozen" on locations. More formally, it is a tuple $\mathcal{A} = (Q, \ell_0, \mathcal{X}, Inv, \Sigma, \mathcal{S}, T)$ with $Q, \ell_0, \mathcal{X}, Inv, \Sigma, T$ have the same definition as in def 1 and $\mathcal{S} : Q \to 2^{\mathcal{X}}$ is a function assigning to a location ℓ the set of stopped clocks at ℓ. The definition for discrete transitions is the same as in def 1, however the time-elapsing transition changes. $(\ell, v) \xrightarrow{\delta} (\ell, v')$: **(1)** elapses some amount of time δ by setting $v'(x) = v(x)$ if $x \in \mathcal{S}(\ell)$, $v(x) + \delta$ otherwise

Reachability of SWA is undecidable in general however, as shown in [19], there exist decidable fragments like **Initialized Stopwatch Timed Automata** (ISWA) for which deciding reachability remains in PSPACE as for TA.

Definition 2 ([19]). ***Initialized Stopwatch Timed Automata*** *a SWA $\mathcal{A}$ is a (ISWA) if for any transition $t = (\ell, g, a, r, \ell')$, if $(x \in \mathcal{S}(\ell) \wedge x \notin \mathcal{S}(\ell'))$ or $(x \notin \mathcal{S}(\ell) \wedge x \in \mathcal{S}(\ell'))$, then $x \in r$.*

That is a clock is only started or stopped if it is also reset. We will show in section 4.2 that our TA abstraction of the running example falls into this category.

Definition 3 ([12]). ***Channel systems*** *are finite automata augmented with a finite number of channels. They can be thought of as FIFO (First In First Out) queues for symbols used for asynchronous communication. During a transition, we can either **(1)** Leave the channels untouched **(2)** Push a symbol into a channel $\mathfrak{c}$, denoted $\ell \xrightarrow{\mathfrak{c}!a} \ell'$ indicating that the symbol a is appended to $\mathfrak{c}$. This is always possible if channels are unbounded, i.e. can contain an unbounded number of symbols. **(3)** Peek and pop a symbol from a channel $\mathfrak{c}$, denoted $\ell \xrightarrow{\mathfrak{c}?a} \ell'$. This action looks at the head of $\mathfrak{c}$. If it contains the symbol a, it is removed from $\mathfrak{c}$ when taking the transition, otherwise the transition is deactivated.*

Bounded channel systems, that is channel systems in which channels can only contain a fixed number of symbols, are decidable. They can be translated into a finite automaton (with exponentially many locations in both the number of different symbols and the maximal length of the channel), which can then form a synchronized product with the other automata.

As a high-level abstraction, we model our example as a parallel composition of ISWA communicating via strong synchronization augmented by channels.

4.2 Timed Automata representation of our running example

To model our car traffic with systems of Timed Automata, we suppose that each car begins/ends at the beginning/end node of a section and that it stops instantly. Let us describe the automata we generate to model our system with a simple example. The full construction is detailed in appendix A.

For each car A, the clock x_A represents its progress along its path. For each directed section $\boldsymbol{s'} = (s'_{[n_i,n_j],L}, \boldsymbol{d})$ of its path, A performs three steps corresponding to three locations in the TA: **(1)** waiting within the section (location $\mathrm{w}_{\boldsymbol{s'}}$); here the associated clock is stopped, **(2)** driving in the section (location $\mathrm{d}_{\boldsymbol{s'}}$), **(3)** arriving at the end of the section (location $\mathrm{a}_{\boldsymbol{s'}}$) after having traveled a distance of L (*resp.* letting L time unit elapses). We represent theses steps in fig 3, in which L_0 denotes the accumulated distance to arrive at the end of section $\boldsymbol{s}$. The timed automaton of car A, is the concatenation of "sub-automata" of each directed section along the path. If $\boldsymbol{s'}$ has no successor (*resp.* predecessor), $\mathrm{a}_{\boldsymbol{s}}$ (*resp.* $\mathrm{d}_{\boldsymbol{s'}}$) is the goal (*resp.* initial) location of the automaton.

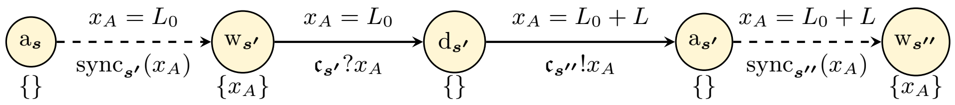

$\mathrm{a}_{\boldsymbol{s}}$
$\{\}$
$x_A = L_0$
$\mathrm{sync}_{\boldsymbol{s'}}(x_A)$
$\mathrm{w}_{\boldsymbol{s'}}$
$\{x_A\}$
$x_A = L_0$
$\mathfrak{c}_{\boldsymbol{s'}}?x_A$
$\mathrm{d}_{\boldsymbol{s'}}$
$\{\}$
$x_A = L_0 + L$
$\mathfrak{c}_{\boldsymbol{s''}}!x_A$
$\mathrm{a}_{\boldsymbol{s'}}$
$\{\}$
$x_A = L_0 + L$
$\mathrm{sync}_{\boldsymbol{s''}}(x_A)$
$\mathrm{w}_{\boldsymbol{s''}}$
$\{x_A\}$

Fig. 3: The sub-automaton of our car Timed Automaton

If a section s is indeed an intersection, then multiple copies of its sub-automaton will be present in the automata of the corresponding cars. In order to ensure that a trace in this abstract model allows for collision avoidance in the real-world, we need to synchronize the different copies. To this end we create the intersection automaton shown in fig 4 for two cars sharing a section in direction $\rightarrow$.

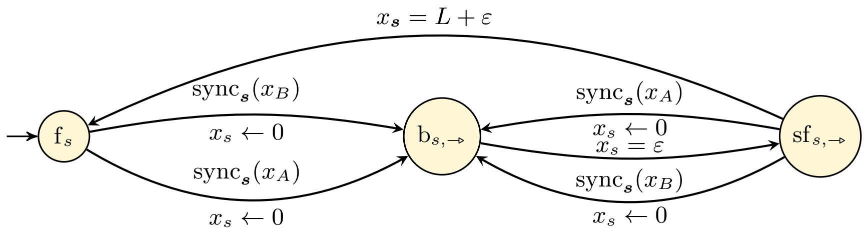

$x_{\boldsymbol{s}} = L + \varepsilon$
f_s
$\mathrm{sync}_{\boldsymbol{s}}(x_B)$
$x_s \leftarrow 0$
$\mathrm{sync}_{\boldsymbol{s}}(x_A)$
$x_s \leftarrow 0$
$\mathrm{b}_{s,\rightarrow}$
$\mathrm{sync}_{\boldsymbol{s}}(x_A)$
$x_s \leftarrow 0$
$x_s = \varepsilon$
$\mathrm{sync}_{\boldsymbol{s}}(x_B)$
$x_s \leftarrow 0$
$\mathrm{sf}_{s,\rightarrow}$

Fig. 4: The intersection automaton of s where A and B can drive in direction $\rightarrow$.

The three states correspond to: **free** (f_s): any car can enter in any direction; **blocked** ($\mathrm{b}_{s,\rightarrow}$): no car can enter; **semi-free** ($\mathrm{sf}_{s,\leftarrow}$): an additional car can enter in $\rightarrow$ direction. Together with the car automaton it ensures that at least ε time units pass before two cars can enter the same section in the same direction, respecting the security distance in an abstract fashion. Similarly, two cars entering the section in opposite direction must be separated by at least $L + \varepsilon$ time units, allowing the first car to cross the section safely before letting the second car in, as shown in the appendix A, page 19, in fig 7. Note that all guards only involve equality testing, which means we could additionally reset the clock to the same value, making our SWA effectively ISWA.

This ensures the safety distance, however not the relative order between the cars: To prevent cars from reversing their order in $\mathrm{w}_{\boldsymbol{s'}}$, cars have to announce themselves on the successor section by pushing their token on the corresponding channel $\mathfrak{c}_{\boldsymbol{s''}}$ before entering $\mathrm{a}_{\boldsymbol{s'}}$. By doing so, the transition from waiting to driving is only activated if the relative order is respected.

4.3 Computing the optimal strategy for reachability

In this section, we present a dedicated algorithm for a system of ISWA synchronized with channels. Several mature tools handling timed automata able to answer reachability problems like IMITATOR ([4]), Uppaal ([7], [8]) Tchecker ([20]) exist. However, at the time of our experiments, none of them supported all the features we needed: channels, stopwatches and time optimality. There are ways to cast our problem as an instance of a more general formulation, but the reachability would then be computed with a general approach that would not benefit from some of the structural specificities of our problem. What would be difficult for a general approach is to detect that a state cannot lead to a solution faster than the best one found so far, and it would typically lead to unnecessarily large computation times. This is what motivated us to propose a specific algorithm for time optimal reachability dedicated to our context.

A time optimal reachability algorithm Our goal is to obtain a time optimal trace witnessing reachability and use it later to construct our control algorithm. We propose a Depth First Search (DFS) forward exploration that uses the properties of our systems of ISWA, synchronized with channels. As we do not use the full expressiveness of parameters, as defined in Parametric Timed Automata ([24], [6] and [5]) (in our model, they never appear in any guards nor are they reset) we can compare the obtained traces and store the current best trace that minimizes the global time. Moreover, we can prune the states during the exploration using a conservative heuristic, leading to significant performance gains.

The full description of our algorithm is presented in appendix B. Here, we give a very brief overview of its principle. States consist of the configuration (location, zone representation via Difference Bounded Matrix (DBM)) and the channels' configuration. We extend the subset relation for zones, denoted $\preceq$, to bounded channels: let s, s' be two states with respective zone z, z', $s \sqsubseteq s'$ iff they have the same location, the same channel configuration and $z \preceq z'$ holds. This gives a partial order on states. As all our guards consist in checking equality, we can compute the successors by letting time elapse in all locations, until a clock matches the equality constraints of the future transition. Finally, we use the conservative heuristic to detect if the current state implies a necessarily larger global time, in which case it is discarded.

5 Ensuring feasibility of high-level plans

Our first layer guarantees correctness in the abstract setting, but is in general not physically realistic/realizable. This gap between the physical reality and the high-level plan has to be closed, or at least bounded, in order to obtain an actually feasible plan. We extract *important* events and their relative order, which represent the *combinatorial* part of the problem, and retain it for the refined solution. We so to speak built upon the high level plan to obtain a more realistic sequence of inputs which still guarantees correctness.

In our running example, the high-level plan is represented by the time optimal SWA-trace. Recall that the control input, the current speed of the car, is represented by the derivative of the clock associated to it. In particular this entails that the car can only be stopped (derivative equals 0) and the car drives at nominal speed (derivative equals 1) and there is no time-delay between them corresponding to infinite acceleration, a hardly realistic supposition.

To this end we rely on satisfiability modulo theories (SMT). The SMT framework provides great flexibility and expressiveness resulting from the wide range of theories and functionalities disposable: Quantifiers, Theory of reals, Minimization of an objective function, arrays and functions etc.

For our running example, using these advanced functionalities comes at a very high cost and we therefore avoid them. We rely on a discretization to allow for a good trade-off between model accuracy and the complexity of the resulting problem. Moreover, instead of minimizing the global time using the built-in minimization functionalities of z3, it has proven to be more effective to perform a linear search, solving the problem for some given maximal global time repeatedly.

From SWA traces to piecewise linear control laws

The SMT instance for traffic control is composed of one set of constraints directly derived from the hybrid system and a second set of constraints representing the high-level plan. We model the traffic system as each car having a piecewise constant speed and discretize time into N steps of equal duration, that we denote δt. We create a variable representing the speed of the ith car during the kth time step, denoted $\tilde{v}_i(k)$, and denote $\tilde{x}_i(k)$ the position, in its path, of the car i at the beginning of the kth timestep. Without loss of generality, we suppose that $\delta t = 1$.

We derive two constraints to guarantee physical realizability under a bounded error. The position is the integral over the velocities, a simple sum as the velocities are piecewise constant: $\tilde{x}_i(k) = \sum_{l=0}^{k-1} \tilde{v}_i(l)$. Secondly we need to bound acceleration and velocity:

$$\forall i,\ \forall k \in [0 \cdots N-2],\ \ \tilde{v}_i(k) - d_{\max} \leq \tilde{v}_i(k+1) \leq \tilde{v}_i(k) + a_{\max}$$
$$\forall i,\ \forall k \in [0 \cdots N-1],\ \ 0 \leq \tilde{v}_i(k) \leq v_{\max}$$

where $d_{\max}$ (*resp.* $a_{\max}$) is the maximal deceleration (*resp.* acceleration) and $v_{\max}$ is the maximal speed of the car ensuring a smaller gap between the actual capabilities of the car and the ones implied by the resulting reference trajectory.

To ensure coherence between the abstract solution represented by the SWA trace and the refined solution represented by the piecewise constant speed, as well as to reduce the search space of the SMT variables, we extract several conditions. Let us denote by $\tilde{p}^0_{i,s}$ (*resp.* $\tilde{p}^1_{i,s}$) the position at which the ith car **enters** (*resp.* **leaves**) some section s, which is obviously only defined if the total trajectory of the ith car passes through section s.

Relative event order We want to ensure that *important* events happen in the same chronological order as given by the SWA trace. *Important* events, in our running example, are the moments when cars enter or leave a section. We impose the constraints for each pair of car and event. Suppose car i enters (*resp.* leaves) the section s before car j enters (*resp.* leaves) section s', this can be imposed by:

$$\forall k \in [0 \cdots N-2], \forall e \in \{0,1\}, \tilde{x}_i(k) < \tilde{p}^e_{i,s} \implies \tilde{x}_j(k) < \tilde{p}^e_{j,s'}$$

Safety distance Whenever a car uses an *intersection*, we need to ensure that it respects a security distance, denoted ε, from the other cars, even if the two cars are not currently sharing a section. Suppose c_i enters section s before c_j, then for any $k \in [0 \cdots N-2]$:

$$(\tilde{p}^0_{i,s} \leq \tilde{x}_i(k) \leq \tilde{p}^1_{i,s} \wedge \tilde{p}^0_{j,s} \leq \tilde{x}_j(k) \leq \tilde{p}^1_{j,s}) \implies ((\tilde{x}_i(k) - \tilde{p}^0_{i,s}) - (\tilde{x}_j(k) - \tilde{p}^0_{j,s}) > \varepsilon)$$

Approximate timing To further restrict the search space for the SMT problem and to increase the similarity between the abstract and refined solution, we do not only keep the relative order between *important* events, but we also impose that they happen at approximately the same global time.

To do this, we introduce a parameter p, which limits the difference between the global time at which an *important* event happens in the high-level plan and the refined plan. This permits a trade-off between the similarity of the high-level, the refined plan and the danger of the SMT instance becoming unsatisfiable (smaller value for p implies higher similarity however the high-level plan may be infeasible for the more realistic model rendering the SMT instance unsatisfiable).

More formally, consider the important event of c_i entering s at the global time t_0. To ensure that the event will actually happens at most p time-units later, we impose: $\tilde{x}_i(t_0 + p) \geq \tilde{p}^0_{i,s}$, since the duration of each step equals 1 (w.l.o.g.).

Interpreting the solution If a satisfying solution for the SMT instance is found, we can readily extract the refined plan from it. All information necessary are the speed values for each time-step and car, *i.e.* the value for all the $\tilde{v}_i(k)$.

6 Reinforcement learning to get real-time policies

Our global approach is divided into 3 stages, and in the previous sections we have presented the 2 first stages, which correspond to distinct levels of abstraction of the multi-agent system we want to control. The solver presented in the two first stages in section 4 and 5 are collectivelly called SMT-SWA solver. Given initial conditions of the system, these two stages enable us to get trajectories for all agents that solve the problem, but not in real-time, so if new initial conditions are faced at a high frequency, and if a high responsiveness is required, then the approach is not practical. For the third stage of our layered approach, which we present in this section, we create a dataset of SWA-SMT solutions on a large number of random instances of the problem, and use this dataset as the initial experience replay buffer of a reinforcement learning (RL) algorithm. Thereby we will first obtain a policy that imitates and slightly generalizes the SWA-SMT solutions, and will then progressively improve. At the end of the learning process, we get a neural network policy that can react in real-time to new conditions and can control the multi-agent system with a high success rate. We could also try to directly solve the multi-agent control problem with RL, with an initially empty experience replay buffer, but with our running example we show that for complex problems, the SWA-SMT solutions are crucial: without the initial dataset, the RL algorithm fails to find any solution to the problem, while with the initial dataset, the RL algorithm quickly matches and then outperforms the success rate of the SWA-SMT approach. In fact, RL algorithms are efficient at progressively improving control policies based on a dense reward signal, but problems with both continuous and combinatorial aspects may result in rewards that are very difficult to find. Multi-agent systems often have these properties, and the associated hard exploration problems are well known failure cases for standard reinforcement learning algorithms [9]. We believe that in this context, a layered approach as the one presented in this paper can be particularly efficient. Using high level abstractions and formal verification, we ignore most of the continuous aspects of the problem, but solve its the most combinatorial and discrete parts, and get traces that can be refined into acceptable solutions. We then build a dataset that can be exploited by reinforcement learning to quickly get good policies, and then iteratively improve them.

To apply reinforcement learning, we cast the problem as a Markov Decision Process (MDP) with a state space S, an action space A, an initial state distribution $p(s_0 \in S)$, a reward function $r(s_t \in S, a_t \in A, s_{t+1} \in S)$ and transition dynamics $p(s_{t+1} \in S | s_t \in S, a_t \in A)$. Since the running example we consider is deterministic, we more specifically use a deterministic transition function: $s_{t+1} = \texttt{step}(s_t, a_t)$. In this MDP, valid SWA-SMT trace should be directly interpretable as high reward episodes. See appendix C for a detailed description of the elements of the MDP for our running example.

Using the initial state distribution, we define random instances of the problem, and run the SWA-SMT solver to get valid solutions, i.e. traces. We then transform each trace into an episode of the MDP. To do so, we first retrieve the sequence of states and actions (see appendix C for details), then we compute the reward for all transitions of the episode, and we terminate the episode if a terminal state is reached (which happens only at the end because we only consider successful traces). The episodes we get correspond exactly to episodes of the MDP previously defined (again, see appendix C for details). We should discard episodes exceeding the maximum number of transitions (85), but we have set this number conservatively so that the time optimal SWA-SMT solutions are always shorter than the limit.

For our running example, we used the SWA-SMT solver on random initial states to generate 2749 successful episodes (with reward ≥ 2000, see appendix C) resulting in a total number of 176913 transitions. About 15% of the random initial conditions were solved by the SWA-SMT solver and led to successful traces.[3]

For the reinforcement learning, we select off-policy algorithms [30] that use a replay buffer to store experience (episode transitions). During training, random batches of transitions are sampled from the buffer, and gradients of loss functions computed on these batches are used to iteratively update the parameters of neural networks (typically the policy network or actor and the value network or critic). New episodes are continuously run with the trained policy to fill the buffer. We compare two approaches, one in which an RL algorithm is trained from scratch (with an initially empty replay buffer), and one in which an RL algorithm starts with its replay buffer filled with the 176913 transitions collected from the SWA-SMT solutions. More specifically, to perform RL form scratch, we first use TD3 [18], a popular off-policy reinforcement learning algorithm, whereas TD3BC [17] is used to perform RL with the replay buffer. TD3BC is TD3 with a slight modification: in the actor loss, a regularization term of behavioral cloning is added, helping the RL algorithm to handle and imitate expert training data that does not come from the trained policy. Originally designed for offline RL (*i.e.* RL on purely offline data, without running episodes), TD3BC can also be seen as a variant of TD3 that is able to start with a non-empty replay buffer initialized with expert data. We use exactly the same hyperparameters for TD3 and TD3BC (see appendix D), and for the additional behavioral cloning regularization term in TD3BC, we use the default parameter $\alpha = 2.5$ (cf. [17]). Fig 5 show the results we obtained with a training of 3 million steps on 5 random seeds for each method[4]. We define successful episodes as episodes with a cumulated reward greater than 2000, which only happens when each car reaches its final destination. We observe that the first approach (TD3 from scratch) never learns to solve the problem. On the other hand, after 250k steps (one step is one discrete time step in an episode play with the neural network policy being trained), the second approach

[3]Generating a successful SWA-SMT trace takes on average about 15sec on a Intel i5-1235u with 16GB of RAM. Note that there is a high variance in this runtime ranging from under a second to several minutes. A timeout was set to 900sec.

[4]Using the *xpag* RL library [29], with a single Intel Core i7 CPU, 32GB of RAM, and a single NVIDIA Quadro P3000 GPU, the training took between 40 and 50 minutes per million steps.

(TD3BC) already reaches the same success rate as the SWA-SMT solver (about 15%), and then it continues to improve during the 3 million steps of training. In the end, we obtain neural network policies with a success rate of approximately 35% in average, which is more than twice the success rate of the SWA-SMT solver. So we not only obtain policies that can be executed in real-time, we also obtain policies that find solutions more consistently.

However, while the SWA-SMT solutions are optimal by construction (*i.e.* they achieve success with the minimum number of steps), there is no such guarantee with the neural network policies trained via reinforcement learning. Fig 6 shows an episode played by a neural network policy trained with TD3BC. A full video of this episode and a few others is hosted at `perso.eleves.ens-rennes.fr/people/Emily.Clement/Implementation/multi-agent.html` The tool we implemented in open-source and can be found at `gitlab.com/Millly/robotic-synthesis`.

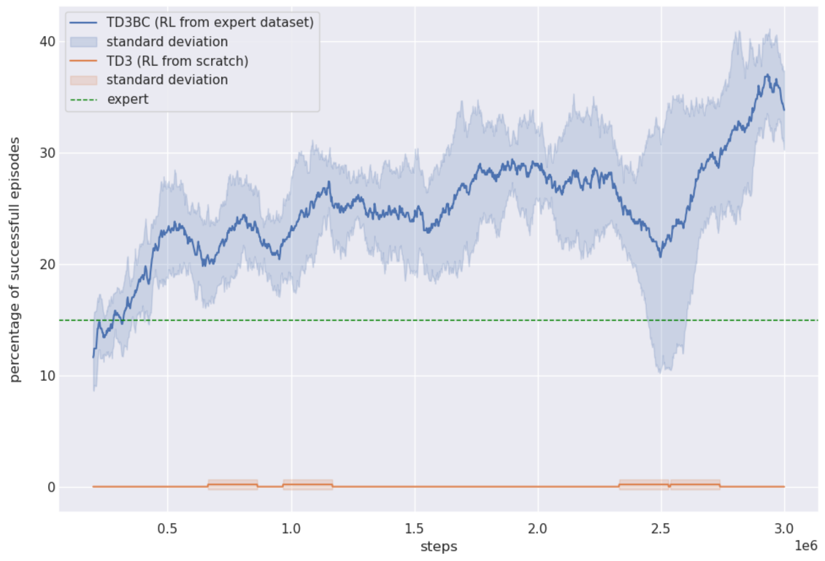


Fig. 5: Percentage of successful episodes during training.

7 Conclusion and future work

We presented a layered approach for multi-agent control involving three stages, the two first ones relying on formal verification tools to compute time optimal solutions, and the last one relying on these solutions (called the SWA-SMT data) to guide a reinforcement learning algorithm. We demonstrated the effectiveness

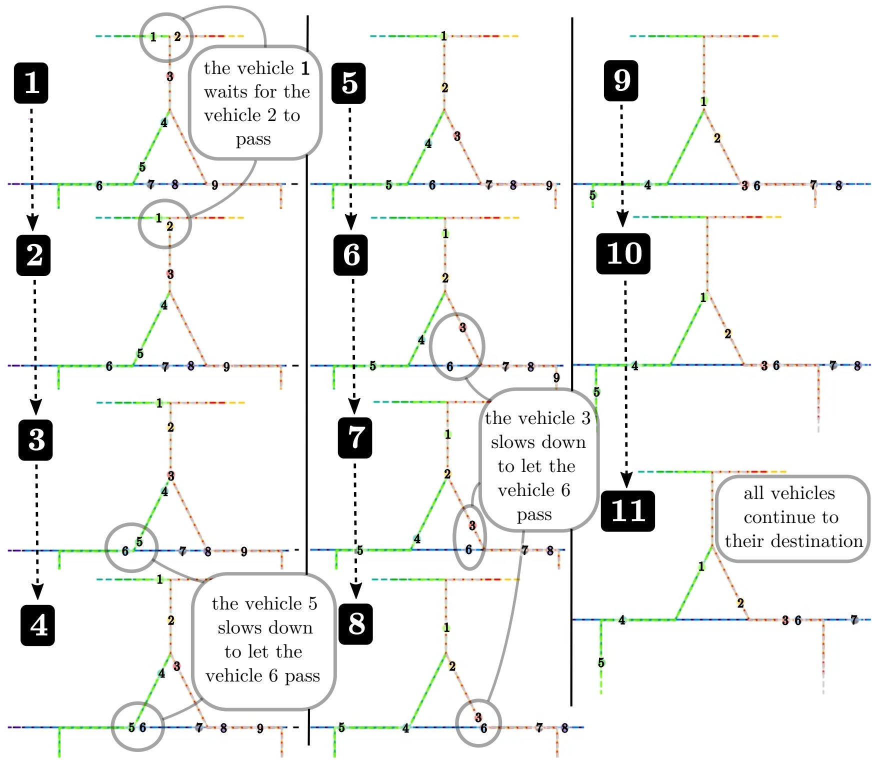

1
2
3
4
5
6
7
8
9
10
11
the vehicle 1 waits for the vehicle 2 to pass
the vehicle 5 slows down to let the vehicle 6 pass
the vehicle 3 slows down to let the vehicle 6 pass
all vehicles continue to their destination

Fig. 6: A successful episode played by a policy trained with TD3BC.

of the approach by applying it to a centralized traffic control problem, showing that, thanks to the SWA-SMT data, the RL algorithm quickly learns to solve the problem and progressively improves to obtain higher success rates. Our results demonstrate the potential of combining layered approaches with RL for multi-agent control. The high-level abstraction in the first stage places emphasis on the combinatorial elements of the problem, leading to high-level plans which are then refined into solutions addressing the actual continuous dynamics of the agents. While it is difficult to construct these solutions in real-time, a rich enough dataset of such solutions can be used to efficiently guide the reinforcement learning, thus eliminating the need for the RL algorithm to tackle the difficult task of exploring the combinatorial aspects of the multi-agent control problem. Ultimately, we obtain a neural network policy capable of controlling the multi-agent system in real-time.

In future work, we would like to implement our proposed algorithm for time optimal reachability in ISWA with bounded channels in the open-source tool TChecker [20], and apply our layered approach to decentralized multi-agent systems.

References

[1] Luis B Almeida. "C1.2 Multilayer perceptrons". In: *Handbook of Neural Computation C* 1 (1997).

[2] R. Alur et al. "The algorithmic analysis of hybrid systems". In: *Theoretical Computer Science*. Hybrid Systems 138.1 (1995), pp. 3–34. ISSN: 0304-3975. DOI: `10.1016/0304-3975(94)00202-T`. URL: `https://www.sciencedirect.com/science/article/pii/030439759400202T` (visited on 03/10/2023).

[3] Rajeev Alur and David L. Dill. "A theory of timed automata". In: *Theoretical Computer Science* 126.2 (1994), pp. 183–235. ISSN: 0304-3975. DOI: `https://doi.org/10.1016/0304-3975(94)90010-8`. URL: `https://www.sciencedirect.com/science/article/pii/0304397594900108`.

[4] Étienne André. "IMITATOR 3: Synthesis of Timing Parameters Beyond Decidability". In: *Computer Aided Verification - 33rd International Conference (CAV 2021)*. Vol. 12759. Lecture Notes in Computer Science. Springer, 2021, pp. 552–565. DOI: `10.1007/978-3-030-81685-8_26`. URL: `https://doi.org/10.1007/978-3-030-81685-8%5C_26`.

[5] Étienne André. "What's decidable about parametric timed automata?" In: *International Journal on Software Tools for Technology Transfer* 21.2 (2019), pp. 203–219.

[6] Étienne André, Didier Lime, and Olivier H Roux. "Decision problems for parametric timed automata". In: *Formal Methods and Software Engineering: 18th International Conference on Formal Engineering Methods, ICFEM 2016, Tokyo, Japan, Proceedings 18*. Springer. 2016, pp. 400–416.

[7] Gerd Behrmann et al. "UPPAAL 4.0". In: *Third International Conference on the Quantitative Evaluation of Systems (QEST 2006), Riverside, California, USA*. IEEE Computer Society, Sept. 2006, pp. 125–126. DOI: `10.1109/QEST.2006.59`. URL: `https://doi.org/10.1109/QEST.2006.59`.

[8] Gerd Behrmann et al. "UPPAAL-Tiga: Time for Playing Games!" In: *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*. Vol. 4590. Lecture Notes in Computer Science. Springer-Verlag, 2007, pp. 121–125.

[9] Marc Bellemare et al. "Unifying Count-Based Exploration and Intrinsic Motivation". In: *Advances in Neural Information Processing Systems*. Vol. 29. 2016. URL: `https://proceedings.neurips.cc/paper_files/paper/2016/file/afda332245e2af431fb7b672a68b659d-Paper.pdf`.

[10] Jur P. van den Berg, Ming C. Lin, and Dinesh Manocha. "Reciprocal Velocity Obstacles for real-time multi-agent navigation". In: *2008 IEEE International Conference on Robotics and Automation, ICRA 2008, Pasadena, USA*. 2008, pp. 1928–1935. DOI: `10.1109/ROBOT.2008.4543489`. URL: `https://doi.org/10.1109/ROBOT.2008.4543489`.

[11] Simon Bøgh et al. "Distributed Fleet Management in Noisy Environments via Model-Predictive Control". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 32. 2022, pp. 565–573.

[12] Daniel Brand and Pitro Zafiropulo. "On Communicating Finite-State Machines". In: *J. ACM* 30.2 (1983), pp. 323–342. DOI: `10.1145/322374.322380`. URL: `https://doi.org/10.1145/322374.322380`.

[13] Jingkai Chen et al. "Scalable and safe multi-agent motion planning with nonlinear dynamics and bounded disturbances". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 13. 2021, pp. 11237–11245.

[14] Alessandro Colombo and Domitilla Del Vecchio. "Efficient algorithms for collision avoidance at intersections". In: *Hybrid Systems: Computation and Control, HSCC'12, Beijing, China*. Ed. by Thao Dang and Ian M. Mitchell. 2012, pp. 145–154. DOI: `10.1145/2185632.2185656`. URL: `https://doi.org/10.1145/2185632.2185656`.

[15] Ali Dorri, Salil S Kanhere, and Raja Jurdak. "Multi-agent systems: A survey". In: *Ieee Access* 6 (2018), pp. 28573–28593.

[16] Paolo Fiorini and Zvi Shiller. "Motion Planning in Dynamic Environments Using Velocity Obstacles". In: *Int. J. Robotics Res.* 17.7 (1998), pp. 760–772. DOI: `10.1177/027836499801700706`. URL: `https://doi.org/10.1177/027836499801700706`.

[17] Scott Fujimoto and Shixiang Shane Gu. "A minimalist approach to offline reinforcement learning". In: *Advances in neural information processing systems* 34 (2021), pp. 20132–20145.

[18] Scott Fujimoto, Herke Hoof, and David Meger. "Addressing function approximation error in actor-critic methods". In: *International conference on machine learning*. PMLR. 2018, pp. 1587–1596.

[19] Thomas A. Henzinger et al. "What's Decidable about Hybrid Automata?" In: *J. Comput. Syst. Sci.* 57.1 (1998), pp. 94–124. DOI: `10.1006/jcss.1998.1581`.

[20] Frédéric Herbreteau and Gerald Point. *The TChecker tool and librairies.* URL: `https://github.com/ticktac-project/tchecker`.

[21] Martin Hilscher, Sven Linker, and Ernst-Rüdiger Olderog. "Proving Safety of Traffic Manoeuvres on Country Roads". In: *Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*. Vol. 8051. Lecture Notes in Computer Science. Springer, 2013, pp. 196–212. DOI: `10.1007/978-3-642-39698-4\_12`. URL: `https://doi.org/10.1007/978-3-642-39698-4%5C_12`.

[22] Martin Hilscher and Maike Schwammberger. "An Abstract Model for Proving Safety of Autonomous Urban Traffic". In: *Theoretical Aspects of Computing - ICTAC 2016 - 13th International Colloquium, Taipei, Taiwan, ROC, Proceedings*. Vol. 9965. Lecture Notes in Computer Science. 2016, pp. 274–292. DOI: `10.1007/978-3-319-46750-4\_16`. URL: `https://doi.org/10.1007/978-3-319-46750-4%5C_16`.

[23] Martin Hilscher et al. "An Abstract Model for Proving Safety of Multi-lane Traffic Manoeuvres". In: *Formal Methods and Software Engineering - 13th International Conference on Formal Engineering Methods, ICFEM 2011, Durham, UK, Proceedings*. Vol. 6991. Lecture Notes in Computer Science.

Springer, 2011, pp. 404–419. DOI: 10.1007/978-3-642-24559-6_28. URL: https://doi.org/10.1007/978-3-642-24559-6%5C_28.

[24] Thomas Hune et al. “Linear parametric model checking of timed automata”. In: *The Journal of Logic and Algebraic Programming* 52 (2002), pp. 183–220.

[25] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization.” In: *ICLR (Poster)*. 2015. URL: http://dblp.uni-trier.de/db/conf/iclr/iclr2015.html#KingmaB14.

[26] Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. “Where’s Waldo? Sensor-Based Temporal Logic Motion Planning”. In: *2007 IEEE International Conference on Robotics and Automation, ICRA 2007, 10-14 April 2007, Roma, Italy*. IEEE, 2007, pp. 3116–3121. DOI: 10.1109/ROBOT.2007.363946. URL: https://doi.org/10.1109/ROBOT.2007.363946.

[27] Xiao Li, Yao Ma, and Calin Belta. “A policy search method for temporal logic specified reinforcement learning tasks”. In: *2018 Annual American Control Conference (ACC)*. IEEE. 2018, pp. 240–245.

[28] Sarah M. Loos and André Platzer. “Safe intersections: At the crossing of hybrid systems and verification”. In: *14th International IEEE Conference on Intelligent Transportation Systems, ITSC 2011, Washington, DC, USA*. IEEE, 2011, pp. 1181–1186. DOI: 10.1109/ITSC.2011.6083138. URL: https://doi.org/10.1109/ITSC.2011.6083138.

[29] Nicolas Perrin-Gilbert. *xpag: a modular reinforcement learning library with JAX agents*. 2022. URL: https://github.com/perrin-isir/xpag.

[30] Doina Precup, Richard S Sutton, and Sanjoy Dasgupta. “Off-policy temporal-difference learning with function approximation”. In: *ICML*. 2001, pp. 417–424.

[31] Arthur Queffelec. “Connected Multi-Agent Path Finding: How Robots Get Away with Texting and Driving.” PhD thesis. University of Rennes, France, 2021. URL: https://tel.archives-ouvertes.fr/tel-03517091.

[32] Roni Stern. “Multi-agent path finding–an overview”. In: *Artificial Intelligence: 5th RAAI Summer School, Dolgoprudny, Russia, July 4–7, 2019, Tutorial Lectures* (2019), pp. 96–115.

A Description of the systems of Timed Automata of for car traffic control

An example

Let us first fully describe the intersection automata of s, presented in section 4.2 if we add a car, called C, where $(s, \leftarrow)$ appears in its path.

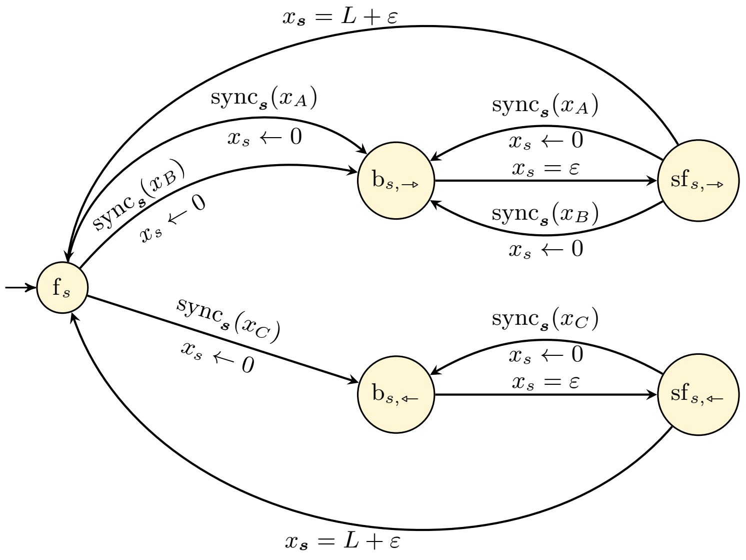


Fig. 7: An intersection automaton of a section s such that A and B can drive in s in direction $\rightarrow$ and C can drive in direction $\leftarrow$.

Construction of the system of ISWA

We have two types of timed automata in our system: automata representing cars and automata representing intersections.

Clocks and parameter We use a parameter, which never appears in any guard, invariant or reset, to represent the *global time*, we denote it t. To describe the progression of each car, we assign, for each car c, a clock, denoted x_c representing c's position along its path. We also assign a clock for each intersection s that we denote x_s. x_s is common for both direction. This clock represents the following value: the value of $p_{\boldsymbol{s}}(c,t)$, where c is the car that is closest to the beginning of the section (whatever direction is used).

Channels To report the presence of a car in a section, each section is assigned a channel. Each time a car wants to enter a section, we push its name as symbol in the channel before entering it, where entering it means accessing the associated waiting state. To be able to drive within the section, we must be able to read its symbol in the channel, that means that this car is the first entered the section among all entered cars. This ensures that a car does not overtake another car.

Automata of cars Let $c = (s_k, \boldsymbol{d}_k)_{0 \leq k \leq m}$ a car of $\mathcal{C}$. We assigned a clock, denoted x_c to this car.

- **Locations:** for each directed section $\boldsymbol{s}$, a car can either *drive* in $\boldsymbol{s}$, *have reached the end* of $\boldsymbol{s}$ and *wait* in $\boldsymbol{s}$. In the last case, we model the waiting state by assigning a stopwatch for x_c.
- **Transitions:** a car can change from waiting to driving in the same section, from driving to arriving and finally from arriving to waiting (to the next section). 1) From waiting to driving, the condition to respect is that x_c is equal to the value of the x_c when entering in $\boldsymbol{s}$. Additionally, we need to respect the order constraint imposed by the channel. 2) From driving to arriving, the clock x_c must have increased by L (length of the section s) and we push the name of the car in the channel dedicated to this section. 3) From arriving to waiting in the next section, the clock must not have increased and need to synchronize with the intersection automaton if the successor section is an intersection.

Automata of intersection section To drive in a section, we have to check beforehand that our car will not collide with another in an abstract section. As each speed is equal between the cars, we can easily check this. For this, the clock associated to the section, x_s is reset each time a new car enters the intersection.

The automaton of s is composed of three or five locations, depending on whether the cars can travel in one or both directions:

- **Blocked Location:** no cars can enter.
- **Semi-free location:** a (unique) car can enter, if it is in the same direction.
- **Free location:** any car can enter, regardless of its direction.

From a Blocked location, we must allow time, equal to the safety distance, to elapse. When this amount of time has elapse, we pass a transition from Blocked to Semi-free location, with the same direction.

Again, from a semi-free location, we have two choices: either another car (with the same direction) enters, reseting the clock and we arrive in a blocked location (with the same direction), or we let time elapse (equals to the length addition with the security distance) and we arrive in Free location.

To indicate a car c enters, the action from semi-free (or free) to blocked, is a synchronized action with the one activated in the automaton of car c, when c enters in the intersection section. Therefore, each automaton of an intersection section has synchronized actions with all cars that will eventually enter in this section.

B Reachability algorithm for ISWA synchronized with channels

Our Depth First Search based algorithm allows us to obtain a time optimal trace for reachability. To do so, we store the fastest trace found so far and compare candidates to this solution during the exploration. Let us describe precisely our algorithm.

We begin by initializing several variables:

– a set of *explored states*, called `explored`. It stores all the states the algorithm has explored yet, in order to avoid exploring a state twice. It is initialized as the empty set.

– `bestSol`= (`bestSolTime`, `bestSolTrace`) a tuple of the total time spent to reach the goal, `bestSolTime`, and the trace of the run, `bestSolTrace`. It represents the best solution found at each step of the algorithm. It is therefore naturally initialized at ($+\infty$, `None`) so that any first trace found be better than it.

– An initial state, s_0 which is a tuple of all the initial configurations (locations and valuations) of the Timed Automata of our system of ISWA.

– a stack, denoted `stack`, which stores pairs of state and the possible successors of this state. The stack is initialized with s_0 and the iterator of the successors of s_0.

In our algorithm, we explore the future states, computed by the function `succ`. If and when we reach the goal state, *i.e.* when all the goal locations of our system of timed automata are reached, we compare the trace of this run with the last stored trace. If the global time is smaller, we replace the stored trace by the trace of our current run.

A conservative heuristic is used to avoid exploring states guaranteed to produce traces that are not time optimal.

Let us detail some very useful sub-functions of our algorithm 1.

• **isFinal:** the helper function `isFinal` determines whether the target location has been reached.

```
$explored \leftarrow \{\}$;                                /* Set of explored states */
$bestSolTrace \leftarrow None$;                            /* Best solution so far */
$bestSolTime \leftarrow \infty$;                           /* Time of bestSol */
$s_0 \leftarrow init()$;                                   /* Initial state */
$stack \leftarrow ()$;                  /* Stack pair(state, successors iterator) */
$stack.push((s_0, succ(s_0, None)))$
while $stack$ is not empty do
    $currState, currSucc \leftarrow stack.pop()$;
    $nextState \leftarrow succ(currState, currSucc)$;
    if $nextState$ is $None$ then
        continue ;                                         /* No more successors */
    end
    $stack.push((currState, nextState))$;
    if $isFinal(nextState)$ then
        if $minGlobTime(nextState) < bestSolTime$ then
            $bestSolTrace \leftarrow traceOf(stack, nextState)$;
            $bestSolTime \leftarrow minGlobTime(nextState)$
        end
        continue;    /* Global time can only grow $\rightarrow$ Ignore successors */
    end
    if $\exists\ s \in explored$ s.t. $nextState \sqsubseteq s$ then
        continue;    /* The subsequent state has already been explored */
    end
    if not $keepExploring(nextState)$ then
        continue;                                  /* Conservative heuristic */
    end
    $stack.push((nextState, None))$;
end
return $bestSol$;
```

Algorithm 1: Time optimal reachability

• **minGlobTime:** the helper function `minGlobTime` extracts the minimal value for the global time with which the target can be reached from the zone.

Since we are only interested in the fastest trace and the global time is never stopped nor reset, we do not need to visit the successor states of an accepting state.

• **keepExploring:** To speed up the algorithm, we use the function `keepExploring` as a heuristic to determine whether it is worth to keep exploring the currently state. In order to ensure correctness of the algorithm, this function needs to be conservative. Meaning that it may never discard a state for which, if the exploration had continued, an accepting trace faster than the currently best solution could be found.

In our running example, this function sums the current global time plus the maximal difference between any current car position and its respective goal position. Intuitively this function computes the time it would take for all cars to arrive if collisions can be ignored from the current situation on. Naturally this heuristic is therefore conservative.

● **succ:** The `succ` function computes, given a current state `currState` and the associated successor iterator `currSucc`, the next successor to explore, denoted `nextState`.

To compute it, we use the special properties of our model.

In our system, described in appendix A, the constraints of our timed automata only contain equality tests of the form $x == x_0$ where x is a clock of the system of timed automata and x_0 is a constant.

Therefore, we do not need to represent zones, but simple valuations contain all information necessary; to compute the delay that elapses before taking the next transition, we just have to compute the minimal delay between all the possibilities.

More precisely, we have these following situations:

– A transition in a car-automaton can be taken.

  If this car is in a waiting location, there are two possibilities: the car takes the transition or it waits until at least another transition is taken in the whole system.

  Otherwise, the transition is taken, as there is no stopwatch.

– In an intersection-automaton, we can be in these three type of locations:

  1. If the location is a Free location, then the choice depends on the car-automaton.
  2. If the location is a Blocked Location, the transition is taken and it "semi-frees" the intersection, as described in appendix A.
  3. If the location is a Semi-free location, two choices are possible: another car asks to enter or the time has elaspe enough to pass to a Free location.

C Markov Decision Process for the running example

The Markov Decision Process is defined by its state space S, action space A, initial state distribution $p(s_0 \in S)$, reward function $r(s_t \in S, a_t \in A, s_{t+1} \in S)$ and deterministic transition function $s_{t+1} = \texttt{step}(s_t, a_t)$.

We describe here all the elements of the MDP defined for our running example:

– **The state space ($\mathbb{R}^{720}$).** The environment contains 3 paths, which are unions of sections, and as detailed in section 2, we have imposed a maximum number of 3 cars per path, so there are at most 9 cars, to which we can attribute a unique identifier (we use $\{-1, 0, 1\}^2$). The state is entirely defined by the speed and position of each car. We could thus use vectors of size 18 to represent states, but instead we chose a sparser representation with a better structure. To remain coherent, we use the same road network presented in sec. 2 which is composed of 24 section (since every car has a dedicated initial and goal node subdividing the sections containing initial and goal positions). On this road network, three different paths are defined, and each section being shared by at most 2 paths. At any given time any section may contain at most 6 cars by construction. For each section, we define a list of 6 tuples, all equal to (0, 0, (0, 0), 0) if no car is currently inside the section. However if there are cars in the section, say 2 cars for example, then the first two tuples have this structure:

$$(\texttt{position with the section}, \texttt{normalized velocity}, \texttt{car identifier}, 1)$$

We represent states as a concatenation of the values of all these tuples for all the 24 sections, which amounts to a vector of size 720. It is a sparse representation, but its advantage is that it makes it easy to find cars close to each other, as they are either in the same section or in neighbor sections.

– **The action space ($\mathbb{R}^9$) and transition dynamics.** Given an ordering of the 9 cars, an action is simply a vector of 9 accelerations. If a_i is the acceleration for the car i, and if at the current time step its position within its path is p_i, and its speed is v_i, then at the next time step its position will be $p_i + v_i$, and its speed will be $v_i + a_i$. This defines the transition dynamics of the MDP. The components of an action corresponding to cars that are not present in the state are simply ignored. Remark: actions can be computed straightforwardly from a sequence of states as they are equal to the difference between consecutive speeds for each car.

– **The reward.** When all cars have reached their destination, i.e. crossed the end of their path, a reward of 2000 is given, and the episode is terminated. Besides, when there is either a collision (a violation of the safety distance between two cars) or two car facing each other in opposite directions in the same section, a negative reward (-100) is given and the episode is terminated. Finally, at each time step, two positive rewards are given, one proportional to the average velocity of the cars (to encourage cars to go fast), and one proportional to the (clamped) minimum distance between all cars (to encourage cars to stay far from each other). We set the maximum number of time step per episode to 85, and adjust these rewards so that an episode cannot reach a cumulated reward of 2000 unless it is truly successful and gets the final +2000 reward.

– **The initial state distribution.** We define an arbitrary initial state distribution in which each of the 9 cars has an 80% chance of being present. The speed of each car is defined randomly, and positions are also defined randomly (within roughly the first two third of each path). Safety distances are ensured, so that the inital states are not in collision, however speeds may be such that there will a collision after the first time step, so there is no guarantee of feasibility.

D Hyperparmeters of the RL algorithms

For TD3:

– Actor network architecture: multi-layer perceptron (MLP) [1] with 3 hidden layers of size 256 and rectified linear unit (ReLU) activation functions.

– Actor optimizer: ADAM [25]

– Actor learning rate: 10^{-3}

– Critic network architecture: MLP with 3 hidden layers of size 256 and ReLU activation functions.

– Critic optimizer: ADAM

– Critic learning rate: 10^{-3}

– Discount factor: 0.99

– Soft update coefficient (τ): 0.05

For TD3BC:

– Actor network architecture: MLP with 3 hidden layers of size 256 and ReLU activation functions.

– Actor optimizer: ADAM

– Actor learning rate: 10^{-3}

– Critic network architecture: MLP with 3 hidden layers of size 256 and ReLU activation functions.

– Critic optimizer: ADAM

– Critic learning rate: 10^{-3}

– Discount factor: 0.99

– Soft update coefficient (τ): 0.05

– α: 2.5