Models for the dynamic exploration of the state spaces of autonomous vehicles.

Johan Arcile„ Raymond Devillers, Hanna Klaudel

Models for the dynamic exploration of the state spaces of autonomous vehicles.

Johan Arcile[1], Raymond Devillers[2], and Hanna Klaudel[1]

[1] IBISC, Univ Evry, Université Paris-Saclay, 91025 Evry, France,
johan.arcile@gmail.com, hanna.klaudel@ibisc.univ-evry.fr
[2] ULB, Bruxelles, Belgium, rdevil@ulb.ac.be

Abstract. We present multi-agent timed models, called MAPTs, where each agent is associated with a regular timed schema upon which all possible actions of the agent rely. MAPTs allow for a layered structure of the state space, so that it is possible to explore the latter dynamically and use heuristics to greatly reduce the computation time needed to address reachability problems.
We then use an available tool for the Petri net implementation of MAPTs, to explore the state space of autonomous vehicle systems and compare this exploration with timed automata-based approaches in terms of expressiveness of available queries and computation time.

1 Introduction

In this paper we propose models and methods tailored for efficient model-checking of multi-agent systems composed of periodic communicating autonomous agents evolving in a constrained environment. Typical examples of such systems are systems of Communicating Autonomous Vehicles (CAVs), where each vehicle (agent) observes its environment (the road, the other vehicles) and periodically evaluates its environment in order to make a decision on the action to perform. The decision policies of CAVs often rely on trajectory planning algorithms studied in 3D simulation, sometimes in conjunction with on-road experiments [18, 19, 14, 20, 15]. While it is possible to plan a collision-free trajectory for a vehicle with this technique, this does not guarantee the efficiency or robustness of a given decision policy.

Therefore, various properties of such systems are of interest to study safety, efficiency and robustness of the decision choices when some variability is introduced in the reaction times, transmission delays, etc. They can be either Boolean ones (*e.g.* Does some vehicle overtake another one before some milestone? Is a collision possible for a given safety distance?) or numerical ones (*e.g.* What is the minimal distance between two vehicles? What is the biggest deceleration for

some vehicle?). This motivates the use of formal methods and several approaches exist focusing on the properties of vehicles (functional layer [13] or soundness of vehicle platooning [17]) or a system of vehicles in their environment. Their drawbacks are twofold: some only allow for basic interactions [26] while others lead to unsatisfactory computation times because of their high-level of realism [22].

The framework VerifCar [5] allows to answer such questions, with a specific focus on the impact of latencies and communication delays on the behaviour of CAVs. Each vehicle is modelled by an agent which performs time restricted actions that impact the valuation of shared variables describing the agents' states and the environment. The system is highly non-deterministic due to overlaps of timed intervals in which the actions of various agents can occur. This model is based on timed automata with an interleaving semantics [2], and is implemented in Uppaal [27], a state of the art tool for real time systems with an efficient state space reduction for model checking. Furthermore, Uppaal allows to consider complex shared data structures and update functions, which are needed to model CAV systems. However, it is limited in terms of the kind of queries that may be asked on the models and deals only with discrete values for the shared variables, which is not always convenient and may lead to imprecise computations.

A closer look at the state spaces in the applications studied with VerifCar shows that they take the form of a directed acyclic graph (DAG). Each agent also has syntactically the form of a DAG between clock resets, and various forms of agents' communications are modelled through shared variables. Our goal is then to exploit these peculiarities to build a dedicated checking environment for reachability problems, when the models satisfy the kind of restricted forms we observed for CAV systems (as well as some other application domains). The hope is that this will allow to get rid of the uncomfortable restrictions we observed on the kind of queries that may be solved, while allowing to do it efficiently. Our aim is thus to change the balance between the expressiveness of the models and the richness of the queries while keeping a reasonable efficiency.

Concretely, we want to explore the state graph dynamically (checking temporal logic properties and computing numerical indicators directly as we explore states) to avoid constructing the full state space, and therefore not to loose time and memory space storing and comparing all previously reached states. The objective is to be able to tune the verification algorithm with heuristics that will choose which path to explore in priority, which might significantly speed up the computation time if the searched state exists. That implies that our algorithms should explore the graph depth-first, since breadth-first algorithms cannot explore paths freely and are restricted to fully explore all the states at some depth before exploring the next one.

For systems featuring a high level of concurrency between actions, such as the CAV systems, most of the non-determinism results from possibly having several actions of different agents available from a given state, that can occur in different orders and which often lead eventually to the same state. This corresponds in the state space to what is sometimes called diamonds. Breadth-first exploration allows to compare states at a given depth and therefore remove duplicates. On

the other hand, depth-first exploring such a state space with diamonds, leads to examine possibly several times the same states or paths, which is not efficient. In this context, our aim is to detect and merge most identical states coming from diamonds while continuing to explore the state space mainly depth-firstly. This diamond detection will consist in a breadth-first exploration in a certain layer of the state space, each layer corresponding to some states at a given depth having common characteristics. It turns out that such layers are typically observed in the state spaces of CAV systems. This allows for a depth-first exploration from layer to layer, while greatly reducing the chances of exploring several times the same states. The class of models on which this kind of algorithm can be applied will be referred to as *Multi-Agent with timed Periodic Tasks* (MAPTs).

In order to analyse a model, it is useful to translate it into formalisms for which we have existing tools. Like in [5], we may use networks of Timed Automata [2] and the tool Uppaal [27] or Time Petri net analyzers, such as Romeo [21] or Tina [7]. However, to keep a better control on the models and queries, we chose to implement our algorithms with Zinc [25], a compiler for high-level Petri nets that generates a library of functions (in Python) allowing to easily explore the state space. In particular, this allows to apply heuristics leading to faster computation times. Another gain when comparing to Uppaal is that it is easy to represent complex data structures, possibly using non-integer variables, thus limiting the loss of information.

The existing works on state space reduction such as [28, 12] use partial-order reductions to explore a subset of the graph, and [8] adapts this methods to Time Petri nets. In those approaches, some independent transitions are merged together to build a single transition in the state space. This requires to know beforehand those different paths leading to a common state, which cannot be done in the models we address. Indeed, diamonds may appear or not depending on the values of the shared variables.

Other approaches, using bounded model checking [11, 10, 23], where a Boolean formula describes an execution of a given length, have been used for studying temporal logic properties. The size of the Boolean formula being exponential in function of the number of variables, this method is effective in terms of computation time only for a restricted length. This is particularly problematic for the systems we study, where the state of the system is described by a large number of scalar variables. In addition, this kind of approaches does not generally allow to prove safety properties, hence to be able to conclude on the non-reachability of a state.

Concerning the exploration algorithms, since the systems we aim to model with MAPTs produce acyclic state spaces, this allows us to use more efficient on-the-fly algorithms for CTL than those proposed in the literature for general cases, such as [9, 6].

In this paper, we start with a formal definition of MAPT, give some examples and provide a translation to high-level Petri nets. Then, we present the exploration algorithms taking advantage of the layered structure, weak and strong variables, and heuristics. Finally, we use MAPT in experiments that highlight

the benefits of our approach in terms of expressiveness of available queries and computation time, and compare them with VerifCar when possible. The contributions of the paper are the introduction and analysis of the MAPTs, as well as the efficient exploration method dedicated to this new formalism.

2 Syntax and semantics of MAPTs

We first define G-MAPTs, composed of several agents that may interact through shared variables. Each agent is associated with a unique local clock and performs actions occurring in some given time intervals. There is no competition between agents in the sense that no agent will ever have to wait for another one's action in order to perform its own actions. However, there may be non-determinism when several actions are available at the same time, as well as choices between actions and time passing.

A G-MAPT is a tuple $(\mathcal{V}, F, A, Init)$ where:

- $\mathcal{V}$ is a set of values;
- F is a (finite) set of calculable functions from $\mathcal{V}$ to $\mathcal{V}$;
- A is a set of n agents such that $\forall i \in [1,n]$, agent $A_i \stackrel{\text{df}}{=} (L_i, C_i, T_i, E_i)$ with:
  - L_i is a set of localities denoted as a list $L_i \stackrel{\text{df}}{=} (l_i^1, \ldots, l_i^{m_i})$ with $m_i > 0$, such that $\forall i \neq j$, $L_i \cap L_j = \emptyset$;
  - C_i is the unique clock of agent A_i, with values in $\mathbb{N}$ (we assume $C_i \neq C_j$ if $i \neq j$, but they may have the same value);
  - T_i is a finite set of transitions, forming a directed acyclic graph between localities, with a unique initial locality l_i^1 and a unique final locality $l_i^{m_i}$, each transition being of the form (l, f, I, l') where $l, l' \in L_i$ are the source and destination localities, $f \in F$ is a transformation function and $I \stackrel{\text{df}}{=} [a, b]$ is an interval with $a, b \in \mathbb{N}$ and $a \leq b$;
  - $E_i \in \mathbb{N} \setminus \{0\}$ is the reset period of agent A_i.
- Init is a triple $(\overrightarrow{\mathsf{init}_L}, \overrightarrow{\mathsf{init}_C}, \mathsf{init}_V) = ((l_1, \cdots, l_n), (\mathsf{init}_1, \cdots, \mathsf{init}_n), \mathsf{init}_V)$ where $\forall i \in [1,n]$, $l_i \in L_i$, $\mathsf{init}_i \in \mathbb{N}$ and $\mathsf{init}_V \in \mathcal{V}$.

For each agent A_i and each locality $l \in L_i$, we shall define by $l^\bullet = \{(l, f, I, l') \in T_i\}$ the set of transitions originated from l, and by ${}^\bullet l = \{(l', f, I, l) \in T_i\}$ the set of transitions leading to l. We assume that ${}^\bullet l_i^1 = \emptyset$ and $(l_i^{m_i})^\bullet = \emptyset$. Moreover, when $i \neq j$, since $L_i \cap L_j = \emptyset$, $T_i \cap T_j = \emptyset$ too, so that each transition belongs to a single agent, avoiding confusions in the model. A simple example of a G-MAPT M with two non-deterministic agents is represented in Ex. 1.

In the semantics, for each agent A_i, we emulate a transition r_i from $l_i^{m_i}$ to l_i^1 that resets clock C_i every E_i time units. As such, each agent in the network cycles over a fixed period. There can be several possible cycles though, since a given locality may be the source of several transitions, so that there may be several paths from l_i^1 to $l_i^{m_i}$. The behaviour of the system is defined as a transition system where Init is the initial state. A state of a G-MAPT composed of n agents is a tuple denoted by $s = (\vec{l}, \vec{c}, v)$ where $\vec{l} = (l_1, \cdots, l_n)$ with $l_i \in L_i$ the

current locality of agent A_i, $\vec{c} = (c_1, \cdots, c_n)$ where $c_i \in \mathbb{N}$ is the value of clock C_i, and $v \in \mathcal{V}$. We shall sometimes consider that v is the value of some shared variable V used to model the evolution of the system.

There are three possible kinds of state changes: a firing of a transition, a clock reset and a time increase. From the state $s = (\vec{l}, \vec{c}, v)$:

- A transition $(l, f, [a, b], l') \in T_i$ can be fired if $l_i = l$ and $a \leq c_i \leq b$. Then, in the new state, $l_i \leftarrow l'$ and $V \leftarrow f(v)$.
- For any agent A_i, clock C_i can be reset if $l_i = l_i^{m_i}$ and $c_i = E_i$. Then, $C_i \leftarrow 0$ and $l_i \leftarrow l_i^1$.
- Time can increase if $\forall i \in [1, n]$, either there exist at least one transition $(l, f, [a, b], l') \in T_i$ with $l_i = l$ and $c_i < b$, or $l_i = l_i^{m_i}$ and $c_i < E_i$. A time increase means that $\forall i \in [1, n]$, $C_i \leftarrow c_i + 1$.

The transition system associated with a G-MAPT is driven by these state changes, from the initial state. It may be observed that there is a single shared element in such a system: variable V; all the other ones are local to an agent. It is unique but its values may have the form of a vector, and an agent may modify several components of this vector through the functions of F used in its transitions, thus emulating the presence of several shared variables. The values of V are not restricted to the integer domain, but there is only a countable set of values that may be reached: the ones that may be obtained from init_V by a recursive application of functions from F (V is not modified by the resets nor the time increases). However this domain may be dense inside the reals, for instance.

Example 1. Let $M = (\mathcal{V}, F, A, \mathsf{Init})$ where:

- $\mathcal{V} = \mathbb{R} \times \mathbb{N}$;
- $F = \{f_1, f_2, f_3, f_4\}$ with $f_1(x, y) \to (2x, y + 1)$ $\quad f_2(x, y) \to (x + 1.3, y + 1)$ $\quad f_3(x, y) \to (\frac{x}{2}, y)$ $\quad f_4(x, y) \to (2x, y)$
- $A = \{(L_1, C_1, T_1, E_1), (L_2, C_2, T_2, E_2)\}$ with
  $L_1 = \{1, 2\}$ $T_1 = \{(1, f_1, [1, 2], 2), (l, f_2, [3, 3], 2)\}$ $E_1 = 5$
  $L_2 = \{3, 4\}$ $T_2 = \{(3, f_3, [1, 2], 4), (3, f_4, [3, 3], 4)\}$ $E_2 = 5$
- $\mathsf{Init} = ((1, 3), (0, 0), (0.5, 0))$.

A visual representation of M and the initial fragment of its dynamics is depicted in Fig. 1. ◊

G-MAPT semantics when considered in a naive way is prone to a state space explosion phenomenon. Motivated by our target applications, we restrict G-MAPTs with Assumption 1 to produce state spaces being DAGs.

Assumption 1 *(acyclicity of the state space)*

1. *$\mathcal{V}$ may be promoted into a partially ordered set $(\mathcal{V}, <)$ and there exist an agent A_i such that, in all paths from l_i^1 to $l_i^{m_i}$ in the local graph of A_i (see for example Fig. 1), there exists a transition $t \stackrel{\text{df}}{=} (l, f, [a, b], l')$ such that for all $v \in \mathcal{V}$, $v < f(v)$; in general, $\mathcal{V} = W \times X$, where $(X, <)$ is a known ordered set and the order is extended in a natural way to $\mathcal{V}$ by stating that $(w, x) < (w', x')$ iff $x < x'$;*

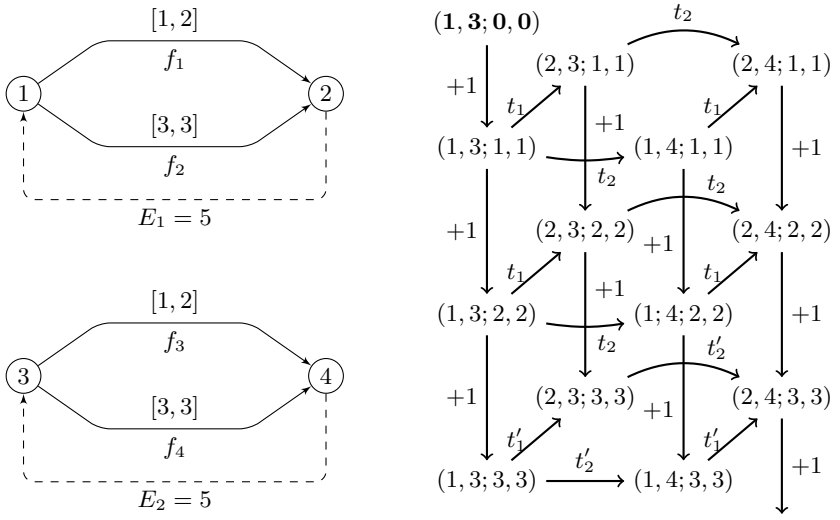


Fig. 1. Left: Visual representation of the two 'local' graphs of G-MAPT from Ex. 1. Right: Initial fragment of the dynamics (without shared variable values).

2. *and there is no* $f \in F$ *such that for some* $v \in \mathcal{V}$*, we have* $f(v) < v$*.*

Item 1 ensures that at least one agent increments the variable V at least once between two of its resets.

Item 2 ensures that no function can decrease the variable V. The result is an absence of cycles in the whole state space of the G-MAPT.

Furthermore, in order to shorten the computation time of the queries, we also consider an accelerated semantics that often allows to increase time by several time units in one step. We shall not detail it here (see [3, 4]) but we shall use it in the experiments shown later in this paper. It needs the following assumptions:

Assumption 2 *(strong liveness)*

1. *If the initial locality of some agent* A_i *is the terminal one* ($l_i = l_i^{m_i}$)*, then the initial value of clock* C_i *satisfies* $\mathsf{init}_i \leq E_i$*;*
2. *otherwise (when* $l_i \neq l_i^{m_i}$*), we have* $\mathsf{init}_i \leq \max\{b \mid (l_i, f, [a, b], l') \in l_i^\bullet\}$*);*
3. *moreover, for each agent* A_i*, if* $l \in L_i \setminus \{l_i^1, l_i^{m_i}\}$*, then* $\max\{b \mid (l', f, [a, b], l) \in {}^\bullet l\} \leq \min\{b' \mid (l, f', [a', b'], l'') \in l^\bullet\}$*;*
4. *and we have* $\max\{b \mid (l', f, [a, b], l_i^{m_i}) \in {}^\bullet l_i^{m_i}\} \leq E_i$*.*

Items 1 and 2 ensure that, when the system is started, either in the terminal locality or in a non terminal one of some agent, the time is not blocked and we shall have the possibility to perform an action in some future.

Item 3 ensures that whenever a non terminal locality is entered, any (and not only some) transition originated from that locality will have the possibility

to occur in some future (the case when we enter an initial locality is irrelevant since resets reinitialise the corresponding clock to 0).

Item 4 captures similar features in the case when we enter a terminal locality. In other words, each transition or reset in $l^\bullet$ is enabled when entering l or will be enabled in the future (after possibly some time passings in order to reach the lower bound a).

Definition 2. *A* MAPT *is a* G-MAPT *satisfying Assumptions 1 and 2.*

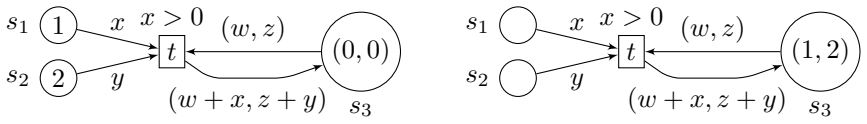
3 Translation into high-level Petri nets

A high-level Petri net [16] can be viewed as an abbreviation of a low-level one [24] where tokens are elements of some set of values that can be checked and updated when transitions are fired[3]. Here, we express a G-MAPT as a high-level Petri net to be implemented with ZINC [25].

Formally, a high-level Petri net is a tuple (S, T, λ, M_0) where:

- S is a finite set of places;
- T is a finite set of transitions;
- λ is a labelling function on places, transitions and arcs such that
  - for each place $s \in S$, $\lambda(s)$ is a set of values defining the type of s,
  - for each transition $t \in T$, $\lambda(t)$ is a Boolean expression with variables and constants defining the guard of t and
  - for each arc $(x, y) \in (S \times T) \cup (T \times S)$, $\lambda(x, y)$ is the annotation of the arc from x to y, driving the production (on y, when $x \in T$ and $y \in S$) or consumption (from x, when $x \in S$ and $y \in T$) of tokens (see an example below).
- M_0 is an initial marking associating tokens to places, according to their types: for each $s \in S$, $M_0(s)$ is a finite multiset of values in $\lambda(s)$.

The semantics of a high-level Petri net is captured by a transition system containing as states all the markings which are reachable from the initial marking M_0. A marking M' is directly reachable from a marking M if there is a transition t enabled at M, whose firing leads to M'; it is reachable from M if there is a sequence of such firings leading to it. A transition t is enabled at some marking M if the tokens in all the input places of t allow to satisfy the flow expressed by the annotations of input arcs and the guard of t, through a valuation of the variables involved in the latter. The firing of t consumes the concerned tokens in input places of t and produces tokens on output places of t, according to the annotations of the output arcs and the same valuation, as shown below for a transition before and after the firing.



[3] We shall allow infinite value sets, but there will never be infinitely many tokens in the system

Definition 3. *Given a* G-MAPT $Q = (\mathcal{V}, F, A, Init)$ *with* $|A| = n$*, its translation to a high-level Petri net* $N = translate(Q) = (S, T, \lambda, M_0)$ *is defined as follows:*

- $S = \{s_A, s_C, s_V\}$ *with* $\lambda(s_A) \stackrel{\text{df}}{=} L_1 \times \cdots \times L_n$ *where* L_i *is the set of localities of agent* A_i *(its* j*th element is denoted* l_i^j*),* $\lambda(s_C) \stackrel{\text{df}}{=} \mathbb{N}^n$ *and* $\lambda(s_V) \stackrel{\text{df}}{=} \mathcal{V}$*. For any token* x *of the type* $\lambda(s_A)$ *or* $\lambda(s_C)$*,* $x[i]$ *is the* i^{th} *element of the list.*
- $T \stackrel{\text{df}}{=} T_{trans} \cup T_{reset} \cup \{t_{time}\}$ *where*
  - T_{trans} *is the smallest set of transitions such that, for each agent* $A_i = (L_i, C_i, T_i, E_i)$ *in* A *and for each transition* $(l, f, [a, b], l') \in T_i$*, there is a transition* $t \in T_{trans}$ *such that* $\lambda(s_A, t) \stackrel{\text{df}}{=} x$*,* $\lambda(s_C, t) \stackrel{\text{df}}{=} y$*,* $\lambda(s_V, t) \stackrel{\text{df}}{=} z$*,* $\lambda(t, s_A) \stackrel{\text{df}}{=} x'$ *where* $x'[i] \leftarrow l'$ *and* $\forall j \neq i$*,* $x'[j] \leftarrow x[j]$*,* $\lambda(t, s_C) \stackrel{\text{df}}{=} y$*,* $\lambda(t, s_V) \stackrel{\text{df}}{=} f(z)$ *and* $\lambda(t) \stackrel{\text{df}}{=} (x[i] = l) \wedge (a \leq y[i] \leq b)$*. This is equivalent to the set of transitions of the* G-MAPT*.*
  - T_{reset} *is the smallest set of transitions such that, for each agent* $A_i = (L_i, C_i, T_i, E_i)$ *in* A*, there is a transition* $t \in T_{reset}$ *such as* $\lambda(s_A, t) \stackrel{\text{df}}{=} x$*,* $\lambda(s_C, t) \stackrel{\text{df}}{=} y$*,* $\lambda(t, s_A) \stackrel{\text{df}}{=} x'$ *where* $x'[i] \leftarrow l_i^1$ *and* $\forall j \neq i$*,* $x'[j] \leftarrow x[j]$*,* $\lambda(t, s_C) \stackrel{\text{df}}{=} y'$ *where* $y'[i] \leftarrow 0$ *and* $\forall j \neq i$*,* $y'[j] \leftarrow y[j]$*, and* $\lambda(t) \stackrel{\text{df}}{=} (x[i] = l_i^{m_i}) \wedge (y[i] = E_i)$ *where* $m_i \stackrel{\text{df}}{=} |L_i|$*. This is equivalent to the set of clock resets of the* G-MAPT*.*
  - $\lambda(s_A, t_{time}) \stackrel{\text{df}}{=} x$*,* $\lambda(s_C, t_{time}) \stackrel{\text{df}}{=} y$*,* $\lambda(t_{time}, s_A) \stackrel{\text{df}}{=} x$*,* $\lambda(t_{time}, s_C) \stackrel{\text{df}}{=} y'$*, where* $\forall i \in [1, n]$*,* $y'[i] \leftarrow y[i] + 1$*, and* $\lambda(t_{time}) \stackrel{\text{df}}{=} G_1 \wedge \cdots \wedge G_n$ *where* G_i *acts as the "upper bound guard" for all the transitions in agent* A_i*, i.e.,* $G_i \stackrel{\text{df}}{=} (g_1 \vee \cdots \vee g_{m_i})$ *with* $m_i = |L_i|$ *and* $\forall j \in [1, m_i - 1]$*,* $g_j \stackrel{\text{df}}{=} (x[i] = l_i^j) \wedge (y[i] < B)$*, where* $B = \max\{b | (l_i^j, f, [a, b], l') \in l_i^{j\bullet}\}$ *is the highest upper bound of the intervals from all outgoing transitions of* l_i^j *and* $g_{m_i} \stackrel{\text{df}}{=} (x[i] = l_i^{m_i}) \wedge (y[i] < E_i)$*. This is equivalent to a time increase.*
- $(M_0(s_A), M_0(s_C), M_0(s_V)) = \mathsf{Init}$ *is the initial marking*

The translation associates singletons as arc annotations for all arcs. As a consequence, during the execution, starting from the initial marking which associates one token to each place, there will always be exactly one token in each of the three places. Each reachable marking, where s_A contains $\vec{l}$, s_C contains $\vec{c}$ and s_V contains v, encodes a state $(\vec{l}, \vec{c}, v)$ of the considered G-MAPT.

It is easy to see [3] that the transition system of a G-MAPT Q is isomorphic to the one of its translated Petri net $translate(Q)$. It is easily possible to modify this translation to correspond to the accelerated semantics (mentioned, but not detailed, above).

4 Dynamic exploration of a MAPT

When model checking a system, one usually has the choice between a depth-first and a breadth-first exploration of the state space. For reachability properties (where one searches if some state satisfying a specific property may be reached),

depth-first (directed and limited by the query) is usually considered more effective. However, the majority of the non-determinism in systems featuring a high level of concurrency (such as CAV systems) leads to diamonds. Indeed, if transitions on different agents are available at a state then they may occur in several possible orders, all of them converging most of the time to the same state (if the applied functions commute). In order to avoid exploring again and again the same states, a depth-first exploration needs to store all the states already visited up to now, which is usually impossible to do in case of large systems. For example, if states $s1$ and $s2$ share a common successor $s3$, when the algorithm will compute the successors of $s1$, it will classically remove $s1$ from the stack and continue with its successors, until reaching $s3$ and exploring all paths from $s3$, forgetting each time the nodes already visited. That way, when the algorithm has explored all paths from $s1$ and starts exploring from $s2$, there is no memory of $s3$ having been explored already, and thus all paths starting from it will be explored again.

On the contrary, using breadth-first algorithms would partly allow to avoid that issue, because duplicate states obtained at a given depth can be removed. However, this would also imply exploring all reachable states at a given depth and forbid using heuristics to guide and limit the exploration.

An idea is then to combine both approaches, which is possible for MAPTs, using the layered approach described next. A MAPT may be non-deterministic but it is strongly live and has a DAG state space. For instance, the G-MAPT M from Ex. 1 satisfies both assumptions (acyclicity is satisfied due to y being incremented in all cycles of A_1) and so is actually a MAPT.

4.1 Layered state space

The state space of a MAPT shows an interesting characteristics: apart from having no cycles, its structure can often be divided in layers such that all states on the border of a layer share the same vectors of localities and clocks (and thus, the same set of enabled transitions) and are situated at the same time distance from the initial state. The only difference concerns the value of the variable, due to the non-determinism and the concurrency inherent to this kind of models. Non-determinism means that an agent has the choice between several transitions at some location; concurrency means that at least two agents may perform transitions at some point. In the first case, several paths may be followed by the agent to reach some point, leading to different values of the variable; in the second case, transitions of the two agents may be commuted, leading again to different values of the variable if the corresponding functions do not commute.

For instance, if agent A_i in a MAPT starts at l_i^1 with a null clock, after E_i time units and before $(E_i + 1)$ time units, it shall necessarily pass through its reset (it is possible that it performs other transitions before and/or after this reset without modifying its clock, but it is sure the agent will go through this reset before performing a new time passing). Hence, if each agent starts from its initial locality with a null clock, after $\mathsf{lcm}\{E_1, \ldots, E_n\}$ (*i.e.*, the least common multiple of the various reset periods; denoted in the following by $\mathsf{lcm}(\mathsf{E})$) time

units, it is sure we shall revisit the initial localities with null clocks, but possibly with other values of the variable V, yielding the border of a layer. From this border the same sequences of transitions/resets/time-passings as initially will occur periodically (with a period of $\mathsf{lcm}(\mathsf{E})$), leading to new borders, with the initial localities and the null clocks.

Exploring layered state space The function $NextBorder(state)$ takes a state $(\vec{l}, \vec{c}, v)$ and computes, through a breadth-first exploration, the set of successors up to the next border. To do so we introduce a function $NextState(s)$, which returns the set of all successors of state s. This is used iteratively in a depth-first exploration to jump from a state to one of its successors belonging to the next border. During this exploration, an additional function may be used to check if a visited state satisfies some condition. Such a use of layers allows to reduce the number of explored paths by detecting diamonds caused by the order of transitions of concurrent agents.

Exploration using strong and weak variables The approach presented above does not deal with diamonds spreading on a time distance longer than the one between two adjacent borders. For example, it may still happen that two different states s_1 and s_2 belonging to the same border have a common successor s in the future. To cope with this issue, it is more interesting to perform the breadth-first exploration that computes successors at the next border for the set $\{s_1, s_2\}$ instead than taking them separately. In general, it is not obvious to know or guess which states should be kept together in the computation of the next border. Indeed, one should be able to determine when sets of states should be split in sub-sets and when they should be kept together. To perform such a clustering, it may be interesting to exploit the properties of target applications, such as CAVs.

A possible solution is to assume $V \stackrel{\text{df}}{=} V_w \times V_s$, where V_w (weak) is a "less important" part of V and V_s (strong) a "more important" one, such that states differing in the valuation of V_s are unlikely to have a common successor, while this is not the case for V_w. Symmetrically, states with the same valuation of V_s are more likely to have a common successor. This may give us a criterion to cluster states and jump from a set of states to the set of their successors at the next border. The choice of V_s and V_w is of course system-dependent and should be defined by an expert, or with the help of a simulation tool. As an example, elements that can be assigned a new value independently of their previous one might be considered as weak, while elements whose value changes depend on their present value (for instance the position of a moving object) might be considered as strong.

The function $ClusteredNextBorder(state\_set)$ is then an easy variant of $NextBorder(state)$, taking a set of states and producing a set of clusters, *i.e.*, sets of states having identical values of variables in V_s. It is used in a similar way as $NextBorder()$ to explore in a depth-first manner the layered state space, the only difference being that it jumps from a cluster belonging to some border to a cluster belonging to the next one, based on the choice of V_s.

4.2 Exploration algorithms

This section is dedicated to exploration algorithms of finite prefixes of MAPTs: states that do not have successors in the considered prefix will be called *final*. The algorithms are denoted with the CTL temporal logic syntax. Since this temporal logic is meant to explore infinite paths, we shall consider that each final state has a self loop.

Our algorithms have two main characteristics: they operate "on-the-fly", which means that they do not store the entire visited state space (but only a cut of it), and they can be tuned with heuristics defining a priority on paths to be explored, that might significantly speed up the computation time if the searched states exist. To do so we rely on the procedure $ClusteredNextBorder()$ mentioned above. Since our algorithms do not store all the states that have been explored, we do not return traces of execution, unlike what is usually proposed by standard temporal logic model checking tools.

Although the general case for checking reachability CTL formulas has not yet been implemented (left for a future work), it can be done easily thanks to the absence of cycles in the state space. Here, we shall present the algorithms used in our experiments. The atomic formulas p can be any Boolean formula using variables or current localities of the state where p is checked. It is also possible to use clock values, but only with the original semantics: the accelerated semantics reduces the state space in a way that preserves the sequences of transitions, but not all the clock values.

The algorithm for EFp (meaning *a reachable state satisfies* p) consists, starting from a stack containing the initial state, in taking the first element s of the stack, returning it if p is true on s, and otherwise adding the result of $ClusteredNextBorder()$ to the stack. The algorithm continues recursively until reaching p or there is no more states to explore in the considered finite prefix. Additionally, we return *true* if p is satisfied by a state between two borders, *i.e.*, during an application of $ClusteredNextBorder()$.

The algorithm for EGp (meaning *there exists a path where* p *is always true*) works in a similar way, but the state s is returned if p is true on s and if s is final, and $ClusteredNextBorder(s)$ is added to the stack only if p is true on s. Additionally, states where p is not true are dropped when explored in $ClusteredNextBorder()$. That way, only states where p is true are explored.

We shall also describe how to check nested CTL queries, of the kind $EF(p \wedge EFq)$, meaning that *a reachable state satisfies* p *and from that state a reachable state satisfies* q, and $EF(p \wedge EGq)$, meaning that *a reachable state satisfies* p *and from that state there exist a path where* q *is always true*. Those nested queries are implemented using a marking function (*i.e.*, a Boolean indicator). $EF(p \wedge EFq)$ is implemented as follows. Whenever p is true on a state, the state is marked. Whenever a state is marked, all its successors are marked. Starting from a stack containing the initial state, the first element s of the stack is returned if q is true on s and s is marked. Otherwise, the result of $ClusteredNextBorder(s)$ is added to the stack. The same marking process is performed between two borders, *i.e.*, in $ClusteredNextBorder()$. We continue recursively until a state

validates the property or there is no more state to explore. As for $EF(p \wedge EGq)$, states are marked whenever both p and q are true or the state is a successor of a marked state and q is true. If a marked final state is reached, it validates the property and is returned. Again, the same marking process is performed in $ClusteredNextBorder()$.

To have a better idea of how those algorithms are implemented, we provide Algorithm 1 computing $EF(p \wedge EGq)$.

Algorithm 1 $EF(p \wedge EGq)$

```
exploring.add(init) {add initial state to the stack of states to explore}
while exploring do
  successors ← NextBorder(exploring.pop()) {removes the state at the end of the
  stack and store all its successors}
  for all s ∈ successors do
    if (check(p, s) or parent_marked(s)) and check(q, s) then
      mark(s) {s is marked if either p is true on s or q is true on s and its prede-
      cessor is marked}
    end if
    if is_marked(s) and check(final, s) then
      return true {returns true if s is a marked final state}
    else if not check(final, s) then
      exploring.add(s) {otherwise and if s is not a final state, add it to the stack}
    end if
  end for
end while
return false
```

5 Experiments

In this section we illustrate the performances of our exploration algorithms applied to three models used in [5], featuring various state space sizes. Those models represent systems of autonomous vehicles circulating on a portion of highway where each vehicle communicates with the other ones to make decisions about its behaviour.

More specifically, we assume that the road section is composed of three unidirectional lanes the vehicles move on, and we store at any time the coordinates (position) of each vehicle on it. Each vehicle is represented by a rectangle with a given length and width, centered on its position. Its state is thus described by a record containing its position, current longitudinal and lateral speeds, longitudinal acceleration (in a given range from negative to positive values), current direction and knowledge about intentions of other vehicles (for example in the form of timed trajectories).

Since we are focusing on the efficiency and robustness of decision choices and not on the control of vehicles (*i.e.*, the module responsible for producing

a trajectory according to the decision choices), we abstract from the physical laws involved in a maneuver such as rotation or inertia. In other words, we assume that turning the steering wheel impacts the speed value that will make the rectangle move on the x-axis, while its longitudinal speed still makes it move on the y-axis.

At a constant frequency of 10 Hz, the speed and position of each vehicle is updated by a transition of an agent *environment update*.

We assume that the vehicles (1) observe the behavior of their neighbors and that the information obtained from the perception sensors is always accurate and immediate and (2) communicate and receive pieces of information, which are not directly observable, such as the planned lane change of the other vehicles; these communications are timed in order to realistically represent the delays between emissions and receptions of data. The behavior of each vehicle consists in repeating endlessly, at its own constant frequency:

- a computation allowing to make a decision on the immediate action to be performed (*i.e.*, execution of the decision algorithm) in the form of an acceleration (speed increase, braking, no change) and a direction (left, right, no change), followed by
- the communication of its own intention (in the form of the trajectory it wishes to follow) to all vehicles (close enough to be) able to receive this information. Received information coming from the other vehicles is stored in the database of the vehicle and used when running the decision algorithm.

A decision algorithm has been implemented for the experiments [3] but should not be considered as one of our contributions: our goal here is not to promote a clever decision making algorithm but to illustrate how MAPTs and their exploration algorithms can be used to efficiently check temporal logic queries in complex multi-agent systems. As such, this section focuses on the comparison of execution times rather than on the actual results, which are detailed and discussed in [5]. Sources for the models and algorithms used in this section are available in [1].

In the following, we first discuss the advantages and drawbacks of using various types of layer-based explorations. Then, we provide some heuristics, and experiment them in order to (hopefully) observe the gain that can be achieved with them. Finally, we compare this method with the framework VerifCar [5], which uses Uppaal, and we provide a verification method for the analysis of such systems that is more efficient and less constrained than the one proposed in [5]. Note that Uppaal uses real clocks but in our context (resets only occur at periodic integer times and functions do not rely on clocks) there is no obstacle in replacing them by integer ones.

5.1 Efficiency of the layer-based exploration.

Here, we compare, for several exploration algorithms, the full exploration time and the reachability time of the first occurrence of a final state. They are explored

in breadth-first, depth-first without layers and depth-first with layers (with and without the use of strong/weak variables). The sizes of the state spaces for Models 1, 2 and 3 are respectively of 7751, 52732 and 285944 states.

Exploration algorithm	Full exploration time (s)			First occurrence of a final state (s)		
	Model 1	Model 2	Model 3	Model 1	Model 2	Model 3
Breadth first	10.8	52.1	420	10.7	52	419.9
Depth-first without layers	∞	∞	∞	3.3	4.6	3.9
Depth-first layered($V_w = \emptyset$)	11	∞	∞	4.5	6.6	4.1
Depth-first layered (small V_w)	11	250	2015	4.5	7.4	6
Depth-first layered (large V_w)	11	71	667	4.5	14.4	7.2

Table 1. Comparison of full exploration time and time to reach the first occurrence of a final state state for exploration algorithms in breadth-first, depth-first with and without layers and with or without the use of weak variables. ∞ means that the exploration was stopped after 50 hours of computation without a result.

As shown in Table 1, one can see that the breadth-first algorithm has the best full exploration time in any case, but the time before reaching any final state is close to the full exploration one, which makes it the worst technique in this case. On the other hand, the standard depth-first algorithm is the fastest for reaching a final state, but it does not fully explore the state space even after 50 hours of computation.

Results for Model 1 show that as long as the layer based approach is used, the full exploration time is very close to that of the breadth-first one. This indicates that there is almost no diamonds covering several layers, meaning that different states belonging to the border of a layer almost never share a common successor. For this reason the use of weak variables has no effect. Although this case is rather simple, it clearly highlights the advantage of layer-based exploration: with almost no increase in full exploration time, it is able to reach a final state much faster.

Model 2 and 3 have much more complex state spaces and, in both cases, the layer-based algorithm that does not rely on weak variables to aggregate states is not able to explore the full state space even after 50 hours of computation. On the contrary, using even a small number but well chosen weak variables (6 out of 39), it is possible to fully explore the state space. In both cases, exploration is about five times longer than the exploration time of breadth-first algorithm. When using a large number of weak variables (30 out of 39), the exploration is much shorter (about 1.5 times the time of breadth-first algorithm). One can note however that the larger V_w, the longer it takes to reach a final state. Indeed, as states are aggregated layer by layer, a too large V_w would result in an exploration similar to a breadth-first one, where all states are kept together and final states are only reached at the end of the computation. With the weak variables chosen, the time to reach a final state remains however reasonable.

In the next experiments, the depth-first algorithms always use layers and a fixed non-empty V_w.

5.2 Heuristics

Exploration algorithms based on layers allow the use of heuristics. These heuristics guide the exploration, choosing among all the unexplored states the one that will most likely lead to a state that satisfies the checked property. The heuristics we use associate a weight to each state, such that when a new state is discovered, it is placed in a list ordered by weight of states to explore. The weight is a prediction of the distance between the current state and a state satisfying the property.

Therefore, a reachability property may be associated with a heuristics that takes a state as an input and returns a weight as an output. Below is a list of heuristics that we used for experiment purposes together with the property they are associated with:

1. *distance_vh1_vh2*: returns the longitudinal position of vehicle vh_1 minus that of vehicle vh_2. It is used with property $EF\ arrival\_vh_1\_before\_vh_2$, where *arrival_vh1_before_vh2* is true in a state if vehicle vh_1 reaches the end of the road portion before vehicle vh_2 does. The idea behind is to check in priority states where vh_1 is the most ahead of vh_2.
2. *estimated_travel_time_vh*: returns the time traveled since the initial state[4] plus the estimated time to reach the end of the road portion, assuming the current speed is maintained. It is used with property $EF\ travel\_time\_vh\_sup\_n$, where *travel_time_vh* $\geq n$ is true in a state if vh reaches the end of the road portion in n time units or more. The idea is to check in priority states where vh is predicted to reach the end of the road with the longest time.
3. *time_to_overtake_vh1_vh2*: is the time before both vehicles arrive at the same longitudinal position if they keep their current speed. It is used with property $EF\ ttc\_vh_1\_vh_2 \leq n$, where *ttc_vh1_vh2* is the value of the time to collision indicator between vh_1 and vh_2 (*i.e.*, the delay before there is a collision between the two vehicles if they keep their current speed), and n is a time to collision value. The idea is to check in priority states where one of the vehicles is getting closer to the other one with the higher speed.

These heuristics have been used on Model 3, with results given in Table 2. The scenario in Model 3 considers three vehicles positioned as depicted in Fig. 2 on a two lane road portion that is 500 m long, with one additional junction lane. Initially, vehicle A is on the right lane at position 0 m with a speed of 30 m/s, vehicle B is on the left lane at position 30 m with a speed of 15 m/s and vehicle C is on the junction lane at position 40 m with a speed of 20 m/s. They all aim at being on the right lane at the end of the road portion.

The first two queries ($EF\ arrival\_B\_before\_A$ and $EF\ travel\_time\_A \geq 15.9$) can only be true in a final state (the deepest layer). As such, the reachability time with the breadth-first algorithm is close to the full exploration time with the same algorithm. In general, the breadth-first reachablity time depends on

[4] Elapsed time since initial state can be estimated by counting how many times an agent has returned to its initial locality.

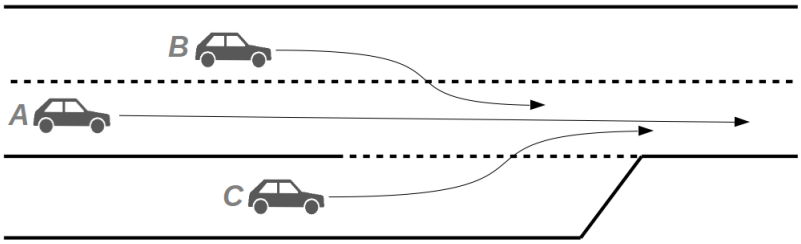


Fig. 2. Initial positions and possible trajectories of autonomous vehicles for the scenario in Model 3.

Exploration algorithm	$EF\ arrival\_B\_before\_A$	$EF\ travel\_time\_A \geq 15.9$	$EF\ ttc\_A\_C \leq 1.14$	$EF\ ttc\_A\_B \leq 0$
Breadth-first	416	427	292	95
Depth-first without heuristics	234—357	167—340	247—547	277—483
Depth-first with heuristics	131	149	103	13

Table 2. Comparison of reachability time for exploration algorithms in breadth-first and depth-first with and without heuristics. As depth-first without heuristics is non deterministic, the two values correspond to the fastest and the slowest runs obtained for each query (five runs where performed each time).

the depth of the first state that satisfies the property. One can observe that for the fourth query ($EF\ ttc\_A\_B \leq 0$), the state is actually reached at a lower depth, which is reflected by the reachability time.

As the depth-first algorithm without heuristics randomly chooses which paths to explore first, the reachability time varies. The number of states in the whole state space that satisfies the property thus impacts the mean reachability time with this algorithm, *i.e.*, when there is more possibility to verify the property, the average time needed is shorter. As we do not want to rely on luck, this is not satisfying.

On the other hand, depth-first algorithm with heuristics explores states in a given order (depending on their weights) and therefore the reachablity time is always the same. The heuristics we used could of course be modified and improved, but they are enough to show a significant decrease of the reachability time. Even for the fourth query, where the breadth-first is faster than the depth-first algorithm, the heuristics allows to quickly identify the state that satisfies the property.

5.3 Comparison with VerifCar

We will now compare the reachability time obtained with Uppaal with the ones obtained with the depth-first exploration algorithms with heuristics, on Model 3. We observed that Uppaal first constructs the state space in about 106 s, then is able to answer almost instantly if a searched state exists. It can therefore answer several queries after constructing the state space, unlike our heuristics-based dynamic exploration algorithms, which have to explore the state space

from scratch for each query. Yet, most of the states we aimed for can be reached easily, and the computation took only about 4 seconds. Queries depicted in Table 2 are those where states were harder to reach. Compared to the ones we obtained in [5], these results indicate that, when a reachability property is satisfied, our algorithms have the same kind of execution time than the ones observed with UPPAAL. On the other hand, if the reachability property is not true, they are slower than UPPAAL, which depending on the kind of query takes between 34 and 370 seconds, which equals the full exploration time with this tool for Model 3. As mentioned previously, the full state space exploration time with depth-first algorithms on this Model, is in our case, of 667 seconds. This is not a surprise since UPPAAL is a mature tool using many efficient optimisations.

However, UPPAAL is restricted in terms of query language, at least in two ways interesting for us. First, it is not possible to directly check bounds of a given numerical indicator, and such bounds should be obtained by dichotomy, requiring several runs for each indicator, such as proposed in the methodology of [5]. Second, it is limited to a subset of CTL (accepting mainly non nested queries). Our algorithms do not have such restrictions.

Indeed it is possible to do a full exploration of the state space while keeping, for each state, the lower and higher values reached on the paths leading to the state, for a given set of indicators. That way, all the information needed to analyse the behaviour of the system, can be obtained after only one full exploration. This is performed as a standard breadth-first exploration (storing states in a file) with the difference that each state is associated with a set of pairs (min, max), being the (temporary) bounds of the considered indicators. Each time a state is explored, the value for each indicator is computed, and it overwrites min if it is smaller, and max if it is greater. That way, each state s contains, for each indicator, the smallest and highest values that exist on the paths from the initial state to s. As several paths can lead to s, we will consider that s reached from path $P1$ and s reached from path $P2$ are equivalent only if the set of their indicators are equal. Therefore, some diamonds might be detected (*i.e.*, two identical states coming from different paths) but not merged together in order to keep information about their respective paths. That way it is possible to have several versions of the same state, but with different indicator values. If one is interested in the reachability of states (for instance, if an indicator is equal to some value), this can easily be done in the same way, by adding Boolean variables to the set of indicators. At the end of the exploration, we get this way the set of all final states, together with all the information that has been carried on their respective paths. It therefore contains all the information needed to analyse finely the system. For the case of Model 3, getting the arrival order together with the bounds for travel time and worst time-to-collision takes 708 seconds. In comparison, the time needed by UPPAAL to obtain the same information with the dichotomy procedure is 3553 seconds.

Also, the DAG shape of the state space allows us to implement nested CTL queries. For the experiments, we used a query of the kind $EF(p \wedge EGq)$, which is the negation of the "leads to" operator $p --> \neg q$ (the only nested operator

available in UPPAAL, in addition to deadlock tests) and two of the kind $EF(p \wedge EFq)$, which cannot be expressed in UPPAAL.

In [5], $arrival\_C\_before\_A \dashrightarrow arrival\_B\_before\_A$ was used and reached a state invaliding the property in 110 seconds. Its negation can be expressed here as $EF(arrival\_C\_before\_A \wedge EG\neg arrival\_B\_before\_A)$ and our algorithm finds the state validating the property in about 10 seconds. The properties
$EF(ttc\_A\_B \leq 1 \wedge EFarrival\_A\_before\_C)$ and
$EF(ttc\_A\_B \leq 1 \wedge EFarrival\_A\_before\_B)$,
that cannot be checked in UPPAAL, can be expressed here. The first one expresses the possibility for vehicle A to arrive ahead of vehicle C after a dangerous situation has occurred, involving a time to collision of less than 1 second. The second is similar for vehicle A and B in the same conditions. The first query is false and needs to explore the whole state space to give an answer (in 680 seconds), while the second one is true and finds a state satisfying the property in about 10 seconds.

Finally, it is worth mentioning that discretisation is needed for UPPAAL, and therefore approximations may be mandatory in some cases, leading to a loss in precision and realism. In addition to a better expressiveness, the model checking process presented in this paper also ensures that no approximation is needed, hence a higher level of realism is achieved.

6 Conclusion

We introduced G-MAPTs, multi-agent timed models where each agent is associated with a regular timed schema upon which all possibles actions of the agent rely. The formalism allows to easily model systems featuring a high level of concurrency between actions, where actions are not temporally deterministic, such as the CAVs. We have then formalised MAPTs (*Multi-Agent with timed Periodic Tasks*), by soundly constraining G-MAPT ones. We also presented how to extract a layered structure out of a MAPT, that allows to detect diamonds while exploring the system depth-first. A translation from G-MAPT to high-level Petri nets allowed us to implement a dedicated checking environment for this formalism with the (free) academic tool ZINC. Algorithms implemented in such environments explore state spaces dynamically and can be used together with heuristics that allow to reduce the computation time needed to reach some states in the model. Finally, experiments highlighted the efficiency of our abstractions, and a comparison of model checking CAVs systems with the framework VERIFCAR has been performed. Although our checking environment does not return traces of executions and is not better for full exploration times than the state of the art tool UPPAAL used in VERIFCAR, it has a better expressiveness both on the model, since we can compute with non-integer numbers, and on the queries since nested CTL ones can be checked. The heuristics performed well for reachability problems and we also provided an exploration algorithm that allows to gather all information needed to analyse the system in one run, which greatly decreased the time needed to gather the same amount of information

when using VerifCar. Although we developed this method with the case study of autonomous vehicles in mind, this formalism and all the abstractions and algorithms presented in this paper can be easily applied to any multi-agent real time systems where agents adopt a cyclic behaviour, such as mobile robots completing cyclically tasks according to their own objectives, flying drone squadrons, etc.

References

1. Sources of the MAPTs models and exploration algorithms. https://forge.ibisc.univevry.fr/jarcile/MAPTs/. Accessed: 2019-10-11.
2. R. Alur and D. Dill. Automata for modelling real-time systems. In *Proceedings ICALP'90*, volume 443 of *LNCS*, pages 322–335. Springer-Verlag, 1990.
3. J. Arcile. *Conception, modélisation et vérification formelle d'un système temps-réel d'agents coopératifs : application aux véhicules autonomes communicants*. PhD thesis, Université Paris-Saclay, France; https://www.biblio.univevry.fr/theses/2019/2019SACLE029.pdf, 2019.
4. J. Arcile, R. Devillers, and H. Klaudel. Dynamic exploration of multi-agent systems with timed periodic tasks. Technical Report: https://hal.archives-ouvertes.fr/hal-02365415, 2019.
5. J. Arcile, R. Devillers, and H. Klaudel. Verifcar: a framework for modeling and model checking communicating autonomous vehicles. *Autonomous Agents and Multi-Agent Systems*, 33(3):353–381, 2019.
6. S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. Scalable distributed on-the-fly symbolic model checking. In *Formal Methods in Computer-Aided Design*, pages 427–441. Springer Berlin Heidelberg, 2000.
7. B. Berthomieu, P.-O. Ribet, and F. Vernadat. The tool TINA – construction of abstract state spaces for Petri nets and time Petri nets. *International Journal of Production Research*, 42(14):2741–2756, 2004.
8. B. Berthomieu and F. Vernadat. State class constructions for branching analysis of time Petri nets. In *In TACAS'03*, page 442–457. Springer-Verlag, 2003.
9. G. Bhat, R. Cleaveland, and O. Grumberg. Efficient on-the-fly model checking for CTL. In *Proceedings of LiCS*, pages 388–397, June 1995.
10. A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu, et al. Bounded model checking. *Advances in computers*, 58(11):117–148, 2003.
11. E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, Jul 2001.
12. E.M. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order reduction. *International Journal on Software Tools for Technology Transfer*, 2:279–287, 1999.
13. M. Foughali, B. Berthomieu, S. Dal Zilio, F. Ingrand, and A. Mallet. Model Checking Real-Time Properties on the Functional Layer of Autonomous Robots. In *International Conference on Formal Engineering Methods (ICFEM 2016)*, Tokyo, Japan, November 2016.
14. A. Furda and L. Vlacic. Enabling safe autonomous driving in real-world city traffic using multiple criteria decision making. *IEEE Intelligent Transportation Systems Magazine*, 3(1):4–17, Spring 2011.

15. S. Glaser, B. Vanholme, S. Mammar, D. Gruyer, and L. Nouveliere. Maneuver-based trajectory planning for highly autonomous vehicles on real road with traffic and driver interaction. *IEEE Transactions on Intelligent Transportation Systems*, 11(3):589–606, 2010.
16. K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use - Volume 1.* EATCS Monographs on TCS. Springer, 1992.
17. M. Kamali, L. A. Dennis, O. McAree, M. Fisher, and S. M. Veres. Formal verification of autonomous vehicle platooning. *Science of Computer Programming*, 148:88 – 106, 2017. Special issue on Automated Verification of Critical Systems (AVoCS 2015).
18. C. Katrakazas, M. Quddus, W.-H. Chen, and L. Deka. Real-time motion planning methods for autonomous on-road driving: State-of-the-art and future research directions. *Transportation Research Part C: Emerging Technologies*, 60:416 – 442, 2015.
19. Y. Kuwata, J. Teo, G. Fiore, S. Karaman, E. Frazzoli, and J. P. How. Real-time motion planning with applications to autonomous urban driving. *IEEE Transactions on Control Systems Technology*, 17(5):1105–1118, 2009.
20. M. Likhachev and D. Ferguson. Planning long dynamically feasible maneuvers for autonomous vehicles. *The International Journal of Robotics Research*, 28(8):933–945, 2009.
21. D. Lime, O. H. Roux, C. Seidner, and L.-M. Traonouez. Romeo: A parametric model-checker for Petri nets with stopwatches. In *TACAS*, pages 54–57. Springer Berlin Heidelberg, 2009.
22. M. O'Kelly, H. Abbas, and R. Mangharam. APEX : Autonomous vehicle plan verification and execution. In *SAE World Congress*, 2016.
23. W. Penczek and A. Lomuscio. Verifying epistemic properties of multi-agent systems via model checking. *Fundamenta Informaticae*, 2(52):167–185, 2003.
24. J. L. Peterson. *Petri Net Theory and the Modelling of Systems.* Prentice Hall, 1981.
25. F. Pommereau. ZINC: a compiler for "any language"-coloured Petri nets. Technical report, IBISC, university of Evry / Paris-Saclay, 2018.
26. M. M. Quottrup, T. Bak, and R. I. Zamanabadi. Multi-robot planning : a timed automata approach. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004*, volume 5, pages 4417–4422 Vol.5, April 2004.
27. Uppaal. http://www.uppaal.org/.
28. F. Vernadat, P. Azéma, and F. Michel. Covering step graph. In *Application and Theory of Petri Nets 1996*, pages 516–535. Springer Berlin Heidelberg, 1996.