Modeling and Control of Heterogeneous Non-Holonomic Input-Constrained Multiagent Systems

Apollon S. Oikonomopoulos and Kostas J. Kyriakopoulos

Abstract—Motivated primarily by the problem of UAV coordination, in this paper we address the problem of coordination of a non-homogeneous group of non-holonomic agents with input constraints. In the first part of the paper, we develop a modeling framework for heterogeneous multi-agent systems that is based on timed automata. To this extent, an appropriate abstraction of the agents’ workspace from our previous works is extended to three-dimensional space, by utilizing hexagonal prisms. The low level agent details are abstracted by virtue of appropriate controllers to motion primitives that can be performed in the individual workspace cells. The resulting models of the non-homogeneous system capture the non-holonomic behavior and the input constraints imposed by the considered systems. In the second part of this paper, we use the developed models in conjunction with formal verification tools to verify the safety and liveness properties of the system, captured by Linear Temporal Logic (LTL) specifications. Using counter-example guided search, we obtain trajectories that satisfy spatio-temporal specifications. Finally, we simulate two case-studies for two and three-dimensional workspaces respectively.

I. INTRODUCTION

This paper extends the authors’ previous work [13] to non-homogeneous groups of agents with varying kinematic capabilities moving in three-dimensional space, through a generalization of the modeling process. The modeling is expanded to include non-homogeneous groups of agents with different kinematic capabilities. The resulting control method remains centralized and open-loop, relying on low-level stabilizing controllers to perform trajectory tracking.

Our main motivation comes from the field of UAV-to-UAV and UAV-to-ground coordination, where the airspace is primarily populated by aircraft of different capabilities (fixed-wing UAVs, helicopters). The inherent complexity of coordinating (heterogeneous) multi-agent systems and the need for safety guarantees has led to the development of a multitude of algorithms, both, in the continuous and in the discrete domain. Fixed-wing aircraft are a special category of agent systems that exhibit a non-holonomic behavior, while possessing restricted actuation capabilities in the sense that their velocities cannot drop below a certain threshold. The proposed framework can handle different types of aircraft assuming the availability of appropriate controllers that can abstract its dynamics to simple motion primitives that capture all those constraints.

A number of approaches can be found in the literature, dealing with safe navigation of multi-agent non-holonomic systems. In the continuous domain, most approaches have been based on potential fields and, especially, navigation functions [15]. Some of the authors of the current work, in [9] introduced a potential field-based approach for multiple holonomic agents, which was extended in [10] to non-holonomic agents.

Apart from the purely continuous approaches, there have been many works in recent years focused on studying multiple agent systems in the context of hybrid systems. In [16], the authors study the safety properties of multiple-aircraft systems modeled as hybrid systems.

With the advent of powerful formal verification tools, there has been interest in taking advantage of the expressive power and verifiability of Temporal Logic propositions for motion planning problems. In [8] the authors use Linear Temporal Logic (LTL) specifications to construct Buchi automata that activate appropriate controllers to carry out a task. These controllers are encapsulated in the so called “primitive tasks modules” and are treated as input-output modules that with appropriate chaining using automated module composition, provide the necessary controllers that are based on multi-robot navigation functions to carry out a complex motion task. The authors of [4] use Temporal Logic formulas to construct high-level motion tasks, while [6] presents a method to convert English language sentences, through Linear Temporal Logic specifications, into high-level motion-planning objectives. In [14], the UppAal model checker [3] was successfully employed to model and verify the operation of a group of holonomic agents under a simple control law. In [7] a reactive scheme is proposed where robot task specifications are described in LTL and based on an environment model encoded in LTL, an automaton is constructed, that is used to activate appropriate controllers that will guarantee that the system will carry out the task. Current research trends in incorporating symbolic methods in robot motion planning are summed up in [2].

The rest of this paper is structured as follows: Section II describes the preliminaries of the problem and the workspace partitioning scheme. Section III covers the transformation of the agent model to a timed automaton suitable for formal verification. Section IV covers the issue of collision detection within the abstract workspace. Section V provides case-studies of the proposed framework’s use, with simulation.
results discussed in section VI. The work concludes with section VII.

II. WORKSPACE MODEL

A. Preliminaries

We consider \( n_a \) spherical agents, and their finite workspace \( \mathcal{W} \subset \mathbb{R}^3 \). The state of each agent is represented by \( \mathbf{x}_i \in \mathcal{W} \times \mathbb{S}^1 \), where \( i \in \{1, \ldots, n_a\} \) and its initial posture is \( \mathbf{x}_i^0 = \mathbf{x}_{i|t=0} \). The dynamics of each agent \( i \) are described by:

\[
\dot{\mathbf{x}}_i = F_i(\mathbf{x}_i, \mathbf{u}_i)
\]

where \( \mathbf{u}_i = [u_i^1, u_i^2, \ldots, u_i^m]^T \) is the vector of control inputs. The control inputs are assumed to be bound by a set of constraints, \( u_i^k_{\text{min}} < u_i^k < u_i^k_{\text{max}}, k = 1, \ldots, m \). We assume a variable \( \tau \in \mathcal{T} \subset [0, \infty) \) called a clock, with \( \tau \equiv 1 \). Furthermore, we assume that the agent may be steered by one or more continuous control laws and that the agent may switch between control laws in pre-determined intervals.

B. Workspace abstraction

We proceed to model the workspace in a discrete manner. In order to guarantee the agent system’s safety, the workspace will be partitioned into non-overlapping cells, and each cell may contain at most one agent at a time. Thus, we will be partitioning \( \mathcal{W} \) into a set of identical cells,

\[
\{\Pi_i\}_i: \bigcup_{i=1}^{n_{\Pi}} \Pi_i = \mathcal{W}
\]

The cells are non-overlapping, i.e. \( \Pi_i \cap \Pi_j = \emptyset \ \forall \ i, j \in \{1, \ldots, n_{\Pi}\} : i \neq j \). This set of cells is said to fully tessellate the workspace \( \mathcal{W} \).

One such tessellation is that created by regular hexagonal prisms in \( \mathbb{R}^3 \), as can be seen in Fig. 1(a), which we will use for the rest of this work. This tessellation is particularly well-suited for avionic applications, because of its symmetry and the layered structure that corresponds to a flight-level model. The choice of hexagonal prisms has the advantage of allowing relatively smooth horizontal movement, based on 60° turns of relatively small curvature, making it more appropriate for agents with bounded trajectory curvature. However, depending on the problem, other tessellations, such as a cubic one, can be used.

We proceed to define some elements of the tessellation used. We assume a set of hexagonal prisms (henceforth “cells”), of side \( a \) and height \( h \), as can be seen in Fig. 1(b). On the center of each face of the cell we define a finite region \( p_i \) called port where \( i \) is the port index. We require that entry into and exit from each cell happens only through these ports. The position of the ports in the center of the cell’s faces ensures that they are aligned with the neighboring cells’ ports. The ports are enumerated according to Fig. 1(b) port index set is defined as \( D = \{0, \ldots, 7\} \).

Any cross-section of the hexagonal honeycomb, parallel to the prism’s hexagonal bases, generates a canvas of non-aligned rows and columns in \( \mathbb{R}^2 \), known as a \( \{6,3\}\)-grid. For each cell, a tuple \( (i, j, k) \in \mathbb{Z}^3 \) describes its position in the honeycomb. \( (i, j) \) denote the prism’s position in the horizontal (x−y) plane, whereas \( k \) indicates the layer number, i.e. the position along the z−axis (see Fig. 2). For every valid cell, the condition \( (i + j) \mod 2 = 0 \) must be satisfied. By convention, we set the center of cell \( (0, 0, 0) \) at the origin of \( \mathcal{W} \). i, j and k increase along x−, y− and z−axis respectively. The position of the center of every other cell, \( (i, j, k) \) mapped onto \( (x, y, z) \in \mathcal{W} \) through the following transformation:

\[
\begin{bmatrix}
x \\
y \\
z
\end{bmatrix} =
\begin{bmatrix}
\frac{2}{a} & 0 & 0 \\
0 & \sqrt{3}a & 0 \\
0 & 0 & h
\end{bmatrix}
\begin{bmatrix}
i \\
j \\
k
\end{bmatrix}
\]

We also define a set of motion primitives. A motion primitive is a high-level abstraction of a trajectory segment that connects a cell’s input port with an output port within the same cell, thus a neighboring cell’s input port. The motion primitives are required to take place within exactly one cell and each motion primitive is associated with the time \( t_m = t_{out} - t_{in} \subset \mathcal{T} \) it takes to perform, where \( t_{in} \) is the entry time and \( t_{out} \) the exit time. Denote as \( p_{in} \) the input port and \( p_{out} \) the exit port.

Consider agent \( i \) and its trajectory \( \mathbf{x}_i(t), t \in [t_{in}, t_{out}] \) through a cell \( \Pi_k \). We require the following:

\[
\begin{align}
\mathbf{s}_i(t_{in}) &\perp p_{in}, \quad p_{in} \in \{0, \ldots, 5\} \quad (1a) \\
\mathbf{s}_i(t_{out}) &\perp p_{out}, \quad p_{out} \in \{0, \ldots, 5\} \quad (1b) \\
\mathbf{x}_i(t_{in}) =\ &\begin{bmatrix}
\frac{h}{a \sqrt{3}} \\
f\sqrt{3}
\end{bmatrix}, \quad p_{in} \in \{6, 7\} \quad (1c) \\
\mathbf{x}_i(t_{out}) =\ &\begin{bmatrix}
\frac{h}{a \sqrt{3}} \\
f\sqrt{3}
\end{bmatrix}, \quad p_{out} \in \{6, 7\} \quad (1d) \\
\mathbf{x}(t) &\in \Pi_k \forall t \in [t_{in}, t_{out}] \quad (1e)
\end{align}
\]
Thus, the complete set of possible motion primitives can be written as: \( M = \{ \{ p_{in}, p_{out} \} \mid p_{in}, p_{out} \in D \} \).

Each agent moving in the workspace will possess its own set of motion primitives, a subset of \( M \), complying with its own constraints. Each agent’s set of admissible motion primitives is known \( a \ priori \) and is fixed for the problem’s life cycle.

### III. DISCRETE AGENT MODELING

#### A. Timed Automaton Model

We now proceed to model each agent as a timed automaton:

\[
A = (\mathcal{N}, \mathcal{C}, \Sigma, E, \text{Inv}, G, R)
\]

where

- \( \mathcal{N} \) is a set of discrete locations
- \( \mathcal{C} \) is a set of real-valued variables called clocks. We denote the set of clock constraints, i.e. atomic propositions with respect to the clocks’ values, \( B(\mathcal{C}) \).
- \( \Sigma \) is a set of discrete events
- \( E \subset \mathcal{N} \times \Sigma \times \mathcal{N} \) is the set of transitions or edges.
- \( \text{Inv} : \mathcal{N} \rightarrow B(\mathcal{C}) \) assigns invariants to each location.
- \( R : E \rightarrow B(\mathcal{C}) \) assigns resets to each transition.
- \( G : E \rightarrow B(\mathcal{C}) \) assigns guards to each transition.

#### B. Discrete locations

The set of discrete locations consists of three subsets:

\[
\mathcal{N} = \{ q_1 \} \cup \mathcal{N}_M \cup \{ q_f \}
\]

- \( q_1 \) is the initial location. In this state, the agent is already in a cell and needs time \( t' \) to get to an (already determined) output port.
- \( q_f \) is the final location where the agent has fulfilled its objective.
- \( \mathcal{N}_M \in 2^\mathcal{M} \) is the set of motion locations, corresponding to the motion primitives defined in Sec. II. Depending on its kinematic capabilities, each agent has a different set of motion locations, corresponding to the motion primitives that are feasible given the agent’s constraints. No assumptions are made about the motion primitives, other than the satisfaction of constraints (1). It should be noted that the feasibility of the motion primitives depends on the agents capabilities in conjunction with the cell dimensions, \( h \) and \( a \). This connection is illustrated in Sec. V and is a basis for determining an appropriate cell size.

#### C. Clocks

Each agent possesses five (5) clock variables:

\[
\mathcal{C} = \{ i, j, k, p_{in}, \tau \}
\]

- \( \{ i, j, k \} \) designates the cell the agent is in at the given time instant. These clocks do not progress on their own, i.e. \( \dot{i} = \dot{j} = \dot{k} = p_{in} \equiv 0 \).
- \( p_{in} \) designates the port from which the agent entered the above cell.
- \( \tau \), with \( \dot{\tau} = 1 \) is a local clock for timekeeping within the agent’s context.

#### D. Transitions

The set of transitions, \( E \), is dependent on the agent’s kinematic capabilities. In general,

\[
E \subset (\{ q_1 \} \times \mathcal{N}_M) \cup (\mathcal{N}_M \times \mathcal{N}_M) \times (\mathcal{N}_M \times \{ q_f \})
\]

#### E. Invariants and Guards

Invariants and Guards are conjunctions of atomic clock constraints, of the form \( x \sim n \) or \( x - y \sim n \), where \( x, y \in \mathcal{C}, \sim \in \{ \leq, <, =, >, \geq \} \), \( n \in \mathbb{N} \).

1) Invariants: Each discrete location in \( \mathcal{N} \) is assigned a (possibly empty) set of Invariants, i.e. constraints that are in effect as long as the system resides in the given discrete location.

- States \( q_m \in \mathcal{N}_M \) are each assigned invariants of the form: \( \text{Inv}(q_m) = \{ t = t_m \} \), where \( t_m \) represents the time required for the motion primitive represented by state \( q_m \), as described in the previous section.
- Transitions \( (q_m, q_n) \in \mathcal{N}_M \times \mathcal{N}_M \) are assigned generic guards of the form: \( G((q_m, q_n)) = \{ t \geq t_m \} \). These guards are objective completion predicates. As an example, in the case we simply want to move to a destination location in the workspace, the guard would be \( G((q_i, q_f)) = \{ i = i_d; j = j_d; k = k_d \} \) where index \( d \) denotes destination.

2) Guards: Each transition in \( E \) is assigned a (possibly empty) set of Guards, i.e. constraints that must be satisfied for the system to take the specified transition.

- Transitions \( (q_m, q_n) \in \mathcal{E}_M \) are assigned guards of the form: \( G((q_m, q_n)) = \{ t = t_m \} \), where \( t_m \) represents the time required for the motion primitive represented by state \( q_m \), as described in the previous section.

#### F. Reset maps

Each transition in \( E \) is also assigned a (possibly empty) set of Resets, i.e. assignment operations of the form \( x := n \), \( x \in \mathcal{C}, n \in \mathbb{R} \) that are effective immediately after the system takes the transition specified.

- Transitions \( (q_m, q_n) \in \mathcal{E}_M \subset \mathcal{N}_M \times \mathcal{N}_M \) manipulate the agent’s clock variables in the following manner:

\[
\tau := 0, \tau \in \mathcal{C} \tag{2}
\]

\[
(i,j,k,p_{in}) := g_m(i,j,k,p_{in},q_m), \quad i,j,k,p_{in} \in \mathcal{C}, q_m \in \mathcal{N}_M \tag{3}
\]

Thus the timekeeping clock \( \tau \) is reset at each transition, while the clock variables \( i,j,k,p_{in} \) used for positioning, are manipulated in accordance with the motion primitive that was performed in the previous state. \( q_m \) is a function that captures the result of the motion primitive represented by \( q_m \) on the discrete workspace abstraction.

- Transitions \( (q_1, q_i) \in \mathcal{E}_I \) initialize the agents clock variables:

\[
\tau := 0, \tau \in \mathcal{C} \tag{4}
\]

\[
i := i_0, \quad j := j_0, \quad k := k_0, \quad p_{in} := p_{in,0} \tag{5}
\]
IV. COLLISION DETECTION

One critical aspect of a multiple agent system is collision detection and avoidance. As stated in Sec. II, the workspace cells are assumed to be subject to exclusive access; each cell may contain at most one agent at a given time instant.

In general, there are two possible collision classes: Two agents occupying the same cell at the same time, or two agents “swapping” cells, i.e. crossing the same boundary simultaneously from opposite directions.

Collision detection can in general be performed in two ways, either by performing all \( \binom{n_a}{2} \) possible inter-agent collision checks, or by using a coverage map of the workspace composed by two sub-maps, one indicating which distinct cells are occupied at a given time instant and one indicating which distinct ports are occupied at a given time instant. These sub-maps can be either arrays with \( n_\pi \) elements, each representing a cell or port respectively, or a hash structure offering faster access (typically \( O(1) \)).

Alternatively, the cells and ports can be encoded as bit-fields, with each bit representing a single cell’s or port’s occupation status respectively, with a value of 1 denoting an occupied cell or port. Collision detection can then be implemented by counting the number, \( s \), of 1-bits (i.e. occupied cells) and comparing them to the number of agents. A collision exists if and only if \( s \neq n_a \). Counting the 1-bits of an integer value can be performed efficiently using a number of population count algorithms, see Appendix.

The implementation of collision detection as a separate automaton is discussed in Sec. VI, together with the implementation of the simulated systems.

V. CASE-STUDIES

We consider two case studies, demonstrating the modeling process and the framework’s flexibility.

A. Two-dimensional agents with constant linear velocity

We assume a system of three agents moving in \( \mathbb{R}^2 \), with unicycle-like kinematics:

\[
\begin{bmatrix}
\dot{x}_i \\
\dot{y}_i \\
\dot{\theta}_i
\end{bmatrix} =
\begin{bmatrix}
\cos \theta_i & \sin \theta_i & 0 \\
-\sin \theta_i & \cos \theta_i & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
\dot{u}_i \\
\dot{\omega}_i
\end{bmatrix}
\]

We also assume the following constraint set:

\[
\begin{align*}
\dot{u}_i^0 &= \text{const.} \\
-\omega_i^{\text{min}} &= \omega_i < \omega_i^{\text{max}}, \quad \omega_i^{\text{max}} > 0
\end{align*}
\]

These kinematics leave the agent with essentially only one control input (the angular velocity \( \omega \)) and capture the behaviour of a large group of systems that would rather be steered than braked.

Because of the agents’ kinematic capabilities, we assume only three possible motion primitives for each agent, corresponding to three discrete states: Straight travel \( q_S \), Left turn \( q_L \) and Right turn \( q_R \).

The motion primitives performed in these states are expressed by the following continuous vector fields, steering the agents’ inputs during the respective discrete states:

\[
F(q_S) = (\omega_i = 0), \quad F(q_L) = (\omega_i = \omega_i^L), \quad F(q_R) = (\omega_i = -\omega_i^L)
\]

where \( \omega_i^L \) satisfies inequality (8). A positive sign denotes counter-clockwise rotation.

In order for all motion primitives to be possible for each agent, we assume a grid of cells with the following dimensions:

\[
a \geq \frac{2}{3} \max \left( \frac{u_i^L}{\omega_i^{\text{max}}} \right)
\]

Eq. (9) ensures that the cells are large enough for the least versatile agent to be able to perform all three motion primitives. Given \( a \), we define \( \omega_i^0 = \frac{2u_i^L}{a} \).

Given prism side \( a \) satisfying (9) and the agents’ constant linear velocities, each state’s residence time is:

- For \( q_S \), \( t_S = \frac{\sqrt{2}a}{u_i^L} \)
- For \( q_R \) and \( q_L \), \( t_R = t_L = \frac{\pi}{3a\omega_i} = \frac{\pi a}{3u_i^L} \)

The system’s objective is to find a collision-free trajectory leading all agents from their initial positions to their goal positions. Thus, each agent has an individual objective of passing through a specified cell in space and after its objective is fulfilled, the agent may be removed from the agent set, or stay in its final position.

The individual agents’ objectives can be stated as:

\[
i_1 = i_1^d \land j_1 = j_1^d \land k_1 = k_1^d \land p_{1i_1} = p_{1i_1}^d
\]

where \( (i_1^d, j_1^d, k_1^d, p_{1i_1}^d) \in W \times D \) is the \( i \)-th agent’s goal port.

Implementation and simulation results are discussed in the next section.

B. 3D-capable agents with complex objectives

For the second case-study, we assume a set of two 3D-capable agents with the same kinematics as in the previous case, extended with the addition of a decoupled motion along the \( z \)-axis.

\[
\begin{bmatrix}
\dot{x}_i \\
\dot{y}_i \\
\dot{z}_i \\
\dot{\theta}_i
\end{bmatrix} =
\begin{bmatrix}
\cos \theta_i & \sin \theta_i & 0 & 0 \\
-\sin \theta_i & \cos \theta_i & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
\dot{u}_i \\
\dot{\omega}_i \\
\dot{\alpha}
\end{bmatrix}
\]

We assume the same constraints regarding \( \dot{u}_i \) and \( \omega_i \), and additionally:

\[
\begin{align*}
u_i^{\text{min}} &< \dot{u}_i < \dot{u}_i^{\text{max}} \\
\omega_i^{\text{min}} &< \dot{\omega}_i < \dot{\omega}_i^{\text{max}} \quad \omega_i^{\text{max}} > 0
\end{align*}
\]

Following the same process as in Sec. V-A, we define 5 motion primitives: Straight travel \( q_S \), Left turn \( q_L \), Right turn \( q_R \), Ascend \( q_A \) and Descend \( q_D \).

These motion primitives could describe passenger aircraft flying at cruising altitude and performing standard maneuvers (straight flight, turns and level changes). We assume \( q_S, q_L \) and \( q_R \) to have the same vector fields as in Sec. V-A. Additionally, we assume \( F(q_A) = \{ \dot{u}_i^A = \dot{u}_i^{A, \text{const}} \} \) and \( F(q_D) = \{ \dot{u}_i^D = \dot{u}_i^{D, \text{const}} \} \).
Thus, the residence times for the motion states become:

\[
\begin{align*}
\text{For agent 1, } & t_S = \sqrt{3} \approx 1.73, \quad t_R = t_L = \frac{\pi}{6} \approx 1.05; \\
\text{For agent 2, } & t_S = \frac{2\sqrt{3}}{3} \approx 1.15, \quad t_R = t_L = \frac{\pi}{6} \approx 0.52; \\
\text{For agent 3, } & t_S = \frac{\sqrt{2}}{2} \approx 0.71, \quad t_R = t_L = \frac{\pi}{6} \approx 0.52.
\end{align*}
\]

Since clock constraints in UppAal may contain only integer values, it is necessary to change the time scale, by accepting a 2-decimal precision and multiplying with 100.

Fig. 3 shows the resulting template of the agent modeled in UppAal. The agent models are accompanied by an additional observer automaton, responsible for collision detection. The full models can be found at [12].

The agents use a common synchronization channel, c, which causes the observer automaton to check for collisions every time a transition is performed. The check location of the observer automaton is marked as “committed”, i.e. the system may spend no time in this location. This allows the collision check to be performed atomically. Furthermore, the collision check is run once for every transition, even if two transitions happen “simultaneously”. Thus the case of two agents exchanging cells is also addressed.

As stated in Sec. V-A, the system’s objective is to drive all agents from their initial positions to their goals. Thus, each agent’s accepting state, done, is guarded by the condition

\[i = i_d \land j = j_d \land p_{in} = p_{in,d}\]

In order to use the counter-example search capabilities of the model checker, UppAal was asked to verify the negated proposition:

\[A[] \text{not (agent1.done } \&\& \text{ agent2.done } \&\& \text{ agent3.done } \&\& \text{ obs.ok)}\]

This proposition implies that there is no execution sequence, such that all agents finally reach their goals while no collision happens. Using \((0,0,0,1), (5,5,0,3), (3,5,0,3)\) and \((5,5,0,0), (0,0,0,4), (5,1,0,2)\) as the agents’ starting and goal positions respectively (in agent order), UppAal proved this proposition to be false and the counter-example trace generated contained a set of collision-free trajectories. The set of trajectories can be seen in Fig. 4.

The run time was about 60 sec on a 2.0 GHz Intel® Xeon CPU. It should be noted that current versions of UppAal are available only as 32-bit binaries, limiting the effective RAM usage (and thus the state space) to 4GB.

B. 3D-capable agents with complex objectives

The agent model of the previous section was extended to include the level changing states, \(q_A\) and \(q_D\). Because during these states an intermediate cell traversal happens as the agent passes through port \(p_p\) of the underlying cell or \(p_c\) of the overlying cell, states \(q_A\) and \(q_D\) are implemented using two sub-states each:

\[\begin{align*}
\text{For } q_{x1}, & \ x \in \{A, D\} \text{ captures the first half of the motion primitive, while the agent is still in the cell of origin.} \\
\text{For } q_{x2}, & \ x \in \{A, D\} \text{ captures the second half of the motion primitive, where the agent moves in the cell above (resp. below) the cell of origin to reach the target cell.}
\end{align*}\]

In addition to the values of the previous section, we assume \(t_{A1}^1 = 2, t_{D1}^1 = 1.8, t_{A1}^2 = 1.4, t_{D1}^2 = 1.2\).

Having modeled the agents, we are able to also include temporal predicates in the system’s task. In this case, we require that both agents, starting from different points, exit

![Fig. 4. Snapshots of the resulting trajectory set from the system in Sec. VI-A at different time instances. The coloured cells indicate the current positions of the agents. The coloured dots indicate the initial positions of the agents.](image-url)
the space from the same port, with agent 1 exiting first and agent 2 exiting later than 15 time units afterwards. Thus we ask UppAal to verify the proposition:

\[
A[]\text{ not (agent1.done and agent2.done) and (agent1.t - agent2.t > 15) and obs.ok}
\]

Using \(4, 6, -1, 3\) and \(2, 0, 0, 3\) as the agents’ starting positions and \(2, 0, 0, 3\) as the exit port for both agents, UppAal proves this proposition to be false and generated a counter-example trace. It should be noted that local timers, \(\text{agent1.t}\) and \(\text{agent2.t}\), are last reset during the last motion primitive, thus the difference \(\text{agent1.t} - \text{agent2.t}\) is the actual difference between the agents satisfying their objective.

This is a queueing scenario that turns up in different occasions, as for example in air traffic management close to airports. The resulting trajectories from UppAal’s counter-example trace can be seen in Fig. 5.

VII. CONCLUSIONS

We have presented a framework for modeling heterogeneous systems of non-holonomic, input-constrained systems as finite automata and using formal verification tools to generate collision-free trajectories and verify safety and liveness properties of the system. The framework’s use was demonstrated through non-trivial numeric simulation data. Future research directions include investigation of reactive control by suitable modeling of inter-agent and agent-environment interactions, as well as the use of weighted timed automata to capture the notion of cost for different actions.

VIII. ACKNOWLEDGEMENTS

Apollon S. Oikonomopoulos and Kostas J. Kyriakopoulos would like to acknowledge the contribution of the European Commission through project iFly.

APPENDIX

A. Collision detection optimization

It is possible to speed up collision detection using efficient bit-counting techniques. Using a bit to represent each cell in the workspace and having its value set to one every time an agent occupies the cell, converts the collision-detection problem to a population count problem; if the number of ones is less than \(n_a\), then there is a collision in the workspace and the current state must be discarded. The population count problem is well-known and there is a variety of relevant efficient algorithms[11].

Furthermore, modern processors offer a built-in population count (POPCNT) instruction, part of the SSE4.2[5]/SSE4a[1] instruction set, that computes the number of 1-bits in a 64-bit word in a single operation.

REFERENCES


