Coordination of Multiple Non-Holonomic Agents with Input Constraints

Apollon S. Oikonomopoulos*, Savvas G. Loizou† and Kostas J. Kyriakopoulos*

*Control Systems Lab, National Technical Univ. of Athens, Greece   †Mech. Eng. Dept., Frederick University, Cyprus

{apoikos,kkyria}@csl.mech.ntua.gr, s.loizou@frederick.ac.cy

Abstract— **In this paper we present a multi-agent coordination algorithm suitable for systems with aircraft-like kinematic constraints. A model of a system of input-constrained non-holonomic agents is constructed, suitable for use with formal verification tools. The agents considered are uniform and have bounded velocities and limited turning capabilities. We demonstrate how a model checker can be used to generate a counterexample trace for such a system, usable as a trajectory that satisfies our safety and liveness requirements.**

I. INTRODUCTION

Multiple agent systems have been a topic of particular importance in the fields of robotics and control theory. The inherent complexity 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. A special category of agent systems are the input-constrained non-holonomic vehicles, e.g. unicycles with bounded linear and angular velocities. These systems represent a number of real-world vehicle systems, including aircraft at cruising altitude and sea vessels.

A number of approaches has been published, dealing with safe navigation of multiple-agent non-holonomic systems. In the continuous domain, most approaches are potential-field-based and, especially, navigation-function-based [1]. Loizou et al. [2] introduced a potential field-based approach for multiple holonomic agents, which was extended in [3] to non-holonomic agents. Although continuous approaches provide solid theoretical proof, they are difficult to implement in real-world situations, where arithmetic precision and computational power are limited. Furthermore, as they yield stabilizing feedback control laws, it is very difficult to capture the notion of time and handle agent systems with low-bounded velocity. Finally, they have inherent difficulty in dealing with complex obstacles that cannot be mapped to classes of simple obstacles via suitable transformations.

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 [4], 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 Language propositions for motion planning problems. In [5] the authors introduce a way to synthesize feedback controllers for multiple mobile agents. The authors of [6] use Temporal Logic formulas to construct high-level motion tasks, while [7] presents a method to convert English language sentences, through Linear Temporal Logic specifications, into high-level motion-planning objectives. In [8], the UppAal model checker [9] was successfully employed to model and verify the operation of a group of holonomic agents under a simple control law. Current research trends in incorporating symbolic methods in robot motion planning are summed up by *Belta et al.* in [10].

In this paper, a discrete model of a system with input-constrained non-holonomic agents is constructed. The system is by construction and due to the kinematic constraints (low and high velocity bounds and bounded angular velocity) known *not* to be generally safe, i.e. inter-agent collisions are possible and sometimes inevitable. The model of the system is used with a model checker to search for *at least one* counterexample, i.e. a set of safe trajectories that converge to the goal and use the counterexample trace provided by the model checker. Furthermore, a simple algorithm is provided to resolve conflicts around an initial set of trajectories obtained by other means, such as a shortest-path graph search. Simulation results are provided, demonstrating the feasibility of the proposed methods.

The rest of this paper is structured as follows: Section II introduces the problem topology. In Section III the model of the agent as a hybrid system is introduced. Section IV describes the partitioning scheme used to abstract the workspace. Subsequently, Section V describes the problem's reduction to the abstract discrete space. Section VI outlines the collision detection algorithm. In Section VII an implementation of the system is described, which is modified in Section VIII to accommodate lower computational complexity. Simulation results are presented in Section IX and the paper concludes with Section X.

II. PRELIMINARIES

We define the set of identical agents, $A = \{a_i | i = 1, 2, \ldots, n\}$. Each agent has a *position*, $\mathbf{x_i} = [x \ y]^T \in \mathcal{W} \subset \mathbb{R}^2$ and an orientation, $\theta \in [0, 2\pi)$, referenced from a fixed coordinate system, where $\mathcal{W}$ is the problem's workspace, assumed to be finite. Each agent has a *starting position*, $\mathbf{x}_s = [x_s \ y_s]$, a starting orientation, θ_s, a *goal position*, $\mathbf{x}_d = [x_d \ y_d]$ and a goal orientation, θ_d.

The agents are planar and non-holonomic, modeled as points in $\mathcal{W}$, with the kinematics of a unicycle:

$$[\dot{x} \quad \dot{y} \quad \dot{\theta}]^T = [u\cos\theta \quad u\sin\theta \quad \omega]^T$$

where the inputs are u, the linear velocity, and ω the angular velocity of the agent.

The agents' kinematics are subject to the following restrictions:

$$0 < u_{min} < u < u_{max} \tag{1}$$

$$-\omega_{max} < \omega < \omega_{max}, \quad \omega_{max} > 0 \tag{2}$$

These constraints apply to a wide variety of vehicle systems, such as aircraft in cruising altitude, sea vessels, heavy vehicles, etc.

Throughout the rest of this paper, a positive sign on angles and angular velocities will denote counter-clockwise revolution.

III. AGENT MODEL

We now proceed to model the agent, using the notation in [11]. Apart from the agent's position, $\mathbf{x}$, and orientation, θ, each agent is also assigned a continuous variable, $\tau \in T \subset \mathbb{R}^+$ called *clock*. Thus, the agent is a hybrid automaton,

$$\mathcal{H} = \{X, X_0, X_F, F, E, Inv, G, R\} \tag{3}$$

where

- $X = Q \times X_C$ is the state space, where $Q = \{q_I, q_S, q_L, q_R, q_F\}$ is the set of discrete states, and $X_C = T \times \mathcal{W} \times [0, \ 2\pi)$ the set of continuous states.
- $X_0 \subseteq X$ is the set of *initial states*
- $X_F \subseteq X$ is the set of *final* or accepting states,
- $F : Q \to X_C$ yields a vector field $F(q, \cdot)$ for each discrete state,
- $E \subseteq Q \times Q$ is the set of discrete transitions
- $Inv : Q \to 2^{X_C}$ yields the *invariant* of each discrete state,
- $G : E \to Q \times 2^{X_C}$ assigns to each transition $(q_1, q_2) \in E$ a *guard* of the form $\{q_1\} \times V$, $V \subseteq Inv(q_1)$. Each transition's guard is the prerequisites for the transition to be *enabled*.
- $R : E \to Q \times 2^{X_C}$ assigns to each transition $(q_1, q_2) \in E$ a *reset* of the form $q_2 \times U$, $U \subseteq Inv(q_2)$.

We will proceed to describe each of these individually.

A. Discrete states

The system comprises 5 discrete states:

q_I Initial state.

q_S Straight-travel state. In this state the agent moves forward along a straight path.

$q_L(q_R)$ Left- (right-) turning state. In this state the agent performs a counter-clockwise (clockwise) turn of constant curvature.

q_F Final state. In this state the agent has reached the goal point.

According to the definitions of Section II, the sets of initial and final states are $X_0 = \{(q_I, 0, (x_s, y_s), \theta_s)\}$ and $X_F = \{(q_F, \cdot, (x_d, y_d), \theta_d)\}$ respectively.
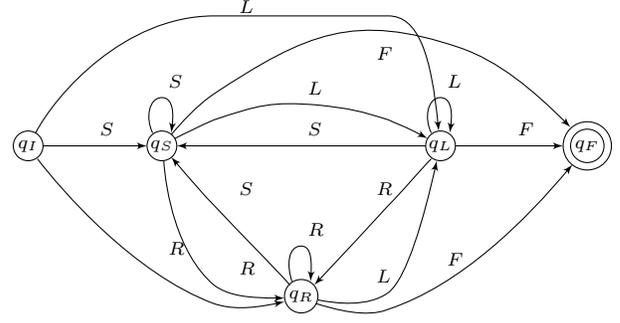
Fig. 1. The hybrid system's discrete transitions. Labels L, S, R and F indicate that the transition leads to q_L, q_S, q_R and q_F respectively.

B. Vector fields

All discrete states assign a vector field that governs the clock's evolution, $\dot{\tau} \triangleq 1$ No assumptions are made about states q_I and q_F, as they are assumed to lie outside the scope of the problem. States q_S, q_L, q_R, each assign one vector field that drives the agent:

$$\dot{\mathbf{p}} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{cases} [u_0\cos\theta \quad u_0\sin\theta \quad 0]^T, & q = q_S \\ [u_1\cos\theta \quad u_1\sin\theta \quad \omega_1]^T, & q = q_L \\ [u_1\cos\theta \quad u_1\sin\theta \quad -\omega_1]^T, & q = q_R \end{cases} \tag{4}$$

We require that the following must hold:

$$u_1 = \frac{\sqrt{3}\pi}{6}u_0, \quad \omega_1 = \frac{2}{3a}u_1 = \frac{\sqrt{3}\pi}{9a}u_0 \tag{5}$$

where a is a parameter discussed in the next section. The choice of u_0 is arbitrary, as long as u_0, u_1 and ω_1 satisfy eqns. (1) and (2). The above choices of u_0, u_1 and ω_1 ensure properties that will be described in the sequel.

C. Transitions

The transitions allowed are depicted in fig. (1). The agent is allowed to turn either left or right, or perform a straight motion, whichever state it might be in, until it reaches its goal.

D. Guards, Resets and Invariants

Each transition $(q_1, q_2) \in E$ has a guard condition on the clock: $G(q_I, \cdot) = \{\tau = 0\}$, $G(q_L, \cdot) = G(q_R, \cdot) = G(q_S, \cdot) = \{\tau \geq 1\}$

The transitions to the final state, q_F, have an additional guard, $G(\cdot, q_F) = \{\mathbf{x} = \mathbf{x}_d \wedge \theta = \theta_d\}$

With every transition a reset of the clock is performed: $R(\cdot, \cdot) = \{\tau := 0\}$. Finally, the following invariants are assigned to the states: $Inv(q_I) = \{\tau \leq 0, \ (x, y) \in \mathcal{W}\}$ $Inv(q_L) = Inv(q_R) = Inv(q_S) = \{\tau \leq 1, \ (x, y) \in \mathcal{W}\}$ The invariant in state q_I forces an immediate transition upon start, whereas the invariants in states q_L, q_R and q_S together with the respective guards force transitions to happen in 1 time-unit intervals.

IV. Workspace partitioning

In this section we will present the partitioning scheme that will map the continuous workspace, $\mathcal{W}$ into a discrete grid.

We partition the euclidean plane in identical, non-overlapping cells, P_i. Such a partitioning, or *tessellation*, is possible in a regular fashion using either triangles, rectangles or hexagons. As will be demonstrated below, the hexagonal regular tessellation offers significant advantages. Thus the workspace cells P_i are regular hexagons of side a, laid out on $\mathcal{W}$ such that:

$$\bigcup_{i=1}^{n} P_i = \mathcal{W}; \ P_i \cap P_j = \emptyset, \ \forall i, j \in \{1, \ldots, n : i \neq j\}$$

We denote the set of cells, $\Pi = \{P_i\}$.
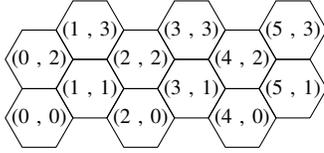
Fig. 2.   Coordinates on the {6,3}-grid

This regular hexagonal tessellation generates a canvas of non-aligned rows and columns, known as a *{6,3}-grid*. Throughout the rest of this paper, the coordinate system depicted in fig. 2 shall be used to identify the cells. For each partition, a tuple $(i, j) \in \mathbb{Z}^2$ describes its position on the grid. Valid cells are only those with $(i + j) \mod 2 = 0$. By convention, we set the center of cell $(0,0)$ at the origin of $\mathcal{W}$. i increases along the x-axis of $\mathcal{W}$ with the same sign and j increases along the y-axis of $\mathcal{W}$ with the same sign. The position of the center of every other cell, (i, j) mapped onto $(x, y) \in \mathcal{W}$ through the following transformation:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \frac{3}{2}a & 0 \\ 0 & \frac{\sqrt{3}}{2}a \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$$

On each cell we define 6 "input ports", one at the center of each side, as depicted in fig. 3. Based on the numbers of the ports, we define the set $\mathcal{D}$ of directions, $\mathcal{D} \triangleq \{0, 1, 2, 3, 4, 5\}$, with each direction being perpendicular to the corresponding port's edge and facing towards the cell's center (fig. 3).

Furthermore, we define the set of motion primitives, $\mathcal{M} \triangleq \{-1, 0, 1\}$, to denote a CCW 60°-arc, straight motion, and a CW 60°-arc respectively, as depicted in fig. 3. Both arcs have a radius of $\frac{3}{2}a$. The values obtained in eqns. (5) by substituting the hexagon side length, a, ensure that all three motion primitives last exactly 1 time unit per clock τ. The choice of hexagon side is arbitrary, as long as eqns. (5) satisfy eqns. (1) and (2) respectively. However, it should be noted
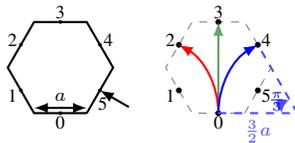
Fig. 3.   The input ports of the hexagonal regions, together with the motions corresponding to a left turn (*red*), straight motion (*green*) and a right turn (*blue*). Note the arrow indicating the *direction* assigned to port 5 on the left.

that the choice of a has a direct impact on the feasibility of the problem: too large a value will cause a coarse partitioning of the workspace, and thus poor space utilization. On the other hand, too low a value will cause a very fine partitioning which will be computationally demanding. Thus, a should be chosen as low as the agent kinematics allow in cluttered environments, where space is scarce, and can be chosen higher in more sparse problems.

This choice of ports, directions and motion primitives has the property that, once on a cell's $d \in \mathcal{D}$ port and taking motion $m \in \mathcal{M}$, the agent will reach another cell's $(d + m) \mod 6$ port with the corresponding direction. This property is modeled by the direction change function $h : \mathcal{D} \times \mathcal{M} \to \mathcal{D}$:

$$h(d, m) = (d + m) \mod 6$$
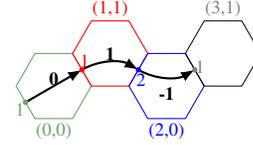
Fig. 4.   Motion on the {6,3}-grid. Colored numbers correspond to the respective cell's input port, p_i, and the numbers in bold indicate the *action* $m_i \in \mathcal{M}$ taken during the corresponding segment. Note that $p_{i+1} = (p_i + m_i) \mod 6$.

Furthermore, we define the cell transition function, $g : \mathbb{Z}^2 \times \mathcal{D} \times \mathcal{M} \to \mathbb{Z}^2 \times \mathcal{D}$:

$$g(i, j, d, m) = \begin{cases} (i, j + 2, 0), & h(d, m) = 0 \\ (i + 1, j + 1, 1), & h(d, m) = 1 \\ (i + 1, j - 1, 2), & h(d, m) = 2 \\ (i, j - 2, 3), & h(d, m) = 3 \\ (i - 1, j - 1, 4), & h(d, m) = 4 \\ (i - 1, j + 1, 5), & h(d, m) = 5 \end{cases} \quad (6)$$

Function g yields the next-step port if an agent is in cell (i, j), port d and takes motion m. The traversal of the workspace grid can be seen in Fig. 4.

Next, we define a trajectory $\mathcal{P}$ as a result of a motion sequence $\mathbf{M} = \{m_i \in \mathcal{M}, \ i = 1, ..., n\}$ applied from a given starting point $\mathbf{X_0} \in \Pi \times \mathcal{D}$ as:

$$\mathcal{P}(\mathbf{X_0}, \mathbf{M}) = \{\mathbf{X_i} : \mathbf{X_i} = g(\mathbf{X_{i-1}}, m_i), \ i = 1, ..., n\}$$

Finally, as can be seen on fig. 4, each port belongs to two adjacent cells. We define the *port identification function*, $r : \mathbb{Z}^2 \times \mathcal{D} \to I$:

$$r(i, j, d) = \begin{cases} (2i, 2j - 2) & \text{if } d = 0 \\ (2i - 1, 2j - 1) & \text{if } d = 1 \\ (2i - 1, 2j + 1) & \text{if } d = 2 \\ (2i, 2j + 2) & \text{if } d = 3 \\ (2i + 1, 2j + 1) & \text{if } d = 4 \\ (2i + 1, 2j - 1) & \text{if } d = 5 \end{cases} \quad (7)$$

where $(i, j) \in \mathbb{Z}^2$ is a valid cell and $d \in \mathcal{D}$ is a port number. This function uniquely identifies each port and will be used for collision detection in the sequel.

V. Reduction to the discrete space

The suitable choice of the vector fields in eq. (4), allows the agent to traverse a cell of the workspace in one clock unit,

$$\tau \geq 1$$
$$(i, j, d) := g(i, j, d, m), \ \tau := 0$$

$$i = i_d \wedge j = j_d \wedge d = d_d$$
$$\tau := 1$$
$$(i, j, d) := (i_s, j_s, d_s)$$
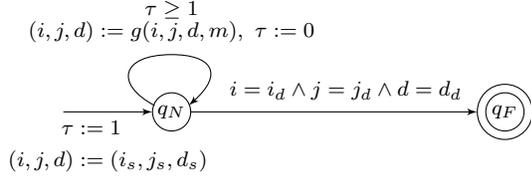
Fig. 5. The timed automaton representing a single agent in the partitioned workspace

$$collision()$$

Fig. 6. The observer automaton checking for collisions in the system. The function $collision()$ is described in section VI.

regardless of the motion primitive performed. We require that all agents start at the same instant from their starting position. We further require that their start and end configurations coincide with cell port positions, and can thus be specified as $(i_s^i, j_s^i, d_s^i) \in \mathbb{Z}^2 \times \mathcal{D}$ and $(i_d^i, j_d^i, d_d^i) \in \mathbb{Z}^2 \times \mathcal{D}$ respectively. This makes all transitions on the grid synchronized and a reduction of the motion in the continuous workspace, to a motion in the discrete grid is possible.

The agent's hybrid automaton can thus be reduced to a timed automaton [12], $\mathcal{A} = \{N, C, \Sigma, \mathcal{B}, l_o, E, I, G, R\}$, where N is a finite set of discrete locations, C is a finite set of real-valued variables called *clocks*, Σ is an alphabet of *events* or *actions*, $\mathcal{B}(C)$ is a set of relations in C called *clock constraints*, l_o is the initial location, $E \subset N \times \mathcal{B}(C) \times \Sigma \times 2^C \times N$ is the set of edges and $I : N \to \mathcal{B}(C)$ is the set of invariants assigned to each location, $G : E \to \mathcal{B}(C)$ is the set of *guards* enabling a transition and $R : E \to 2^C$ is the reset map of the transitions.

We shall be using four (4) clock variables, i, j, d, τ. The first three, i, j, d will be used to designate the cell and the port the agent is currently at. τ will be used as a wall clock. The laws governing the evolution of i, j, d and τ, are $\dot{i} = \dot{j} = \dot{d} \triangleq 0$ and $\dot{\tau} \triangleq 1$, respectively.

Our system has only two discrete states, or *locations*, q_N or *moving* denoting normal motion, and q_F or *done* denoting that the agent has reached its target. The input alphabet, Σ, consists of 3 events, $\{-1, 0, 1\}$, denoting a left turn, straight motion and right turn respectively, as in $\mathcal{M}$. The system's locations, guards and transitions can be seen in fig. 5. The set of guards is:

$$G(q_N, q_F) = \{i = i_d \wedge j = j_d \wedge d = d_d\}$$
$$G(q_N, q_N) = \{\tau = 1\}$$

and the set of resets:

$$R(q_N, q_N) = \{(i, j, d) := g(i, j, d, m)\}$$
$$R(q_N, q_F) = \{\}$$

where $m \in \Sigma$ the current event and $g(\cdot)$ the cell transition function (6). Finally, the set of invariants, is:

$$I(q_N) = \{\tau \leq 1, \ (i, j) \in \Pi\}$$
$$I(q_F) = \{\}$$

Thus the multi-agent system can be formed as a network of timed automata, $\{\mathcal{A}_i\}$, with their time clocks, τ_i synchronized against a wall clock, τ.

VI. COLLISION DETECTION

We will now proceed to define the collision detection algorithm. Given the partitioning of the workspace, there are two classes of possible collisions. Two agents may either traverse the same cell at the same time interval, which is viewed as a collision, or end up on the same *port* coming from the two adjacent cells. These collision classes can be detected using the following algorithm:

```
1  function collision(agents)
2      cells := {}
3      nodes := {}
4      for agent in agents:
5          if agent.position in keys(cells):
6              return true
7          else:
8              cells[agent.position] := 1
9          if agent.node in keys(nodes):
10             return true
11         else:
12             nodes[agent.node] := 1
13     return false
```

In this context, *cells* and *nodes* are hash-like data-structures. The lookup and insert operations on a hash approach $O(1)$ worst-case run-time. This implies that the above version of the collision-detection algorithm approaches $O(n)$ worst-case run-time. This is especially important since collision detection has to take place in every state. Furthermore, the hash-structure approach allows for trivially including any number of stationary obstacles, by pre-populating the *cells* hash with the cells occupied by stationary obstacles, without increasing the overall complexity of the problem.

In order to implement collision detection, an additional automaton, $\mathcal{O}$, with two states is required, which will act as an observer, accessing the agents' clock values. This automaton is shown in fig. 6. The $collision()$ function serves as a guard for the transition to the *bad* state, and may be any of the two variants of the collision-detection algorithm described above.

We can now formally state the problem:
"Given a set of n timed automata, $\{agent_i\}$ and an observer automaton, $collision$, as described above, the system is safe and live, if the following CTL proposition holds:

$$E\left[collision.ok \ \mathbf{U} \ \left(\bigwedge_i agent_i.done \right) \right] \quad (8)$$

i.e. there is a set of collision-free paths that leads to the convergence of all agents to their respective goals, or, equivalently, there are n input sequences that drive all agents to their goals without inter-agent collisions."

VII. IMPLEMENTATION

We are now in position to formally verify our system. We will be exploiting most model checkers' ability to generate traces of counterexamples in case a checked specification

fails. Our implementation is based on the *NuSMV* model checker [13], which was selected primarily for its capabilities and its easy integration with third-party software. Since the agents' transitions are synchronized and the timers are used to store the agents' positions on the grid, the problem is within the modeling capabilities of NuSMV.

Using external, custom-developed software we generate the description of the system and feed it to NuSMV. Subsequently, NuSMV is asked to verify the *negated* version of proposition (8):

$$\neg\mathbf{E}\left[\;collision.ok\;\mathbf{U}\;\left(\bigwedge_i agent_i.done\right)\;\right] \quad\quad (9)$$

i.e., there is *no* trajectory such that (eventually) the system remains collision-free until every agent has reached its goal.

The model checker evaluates the proposition and returns a trace of a counterexample if it finds one. Since the search on the state-space is exhaustive, if there is a solution in the designated state-space, it will be found. The results of simulation runs are discussed in section IX.

VIII. MODEL CHECKING AS A LOCAL SOLVER

Although model checking systems with CTL specifications is a decidable process, in some cases it may be desirable to trade completeness for reduced complexity and use a model checker as a local solver. In this section we demonstrate a simple incremental algorithm that restricts the state-space around a trivial set of trajectories, thus yielding faster results. However, the algorithm presented in this section is not complete. It may find a solution, but not finding one does not imply one does not exist. In order to assure completeness, a full state-space check must be performed, as described in the previous section.

We will be obtaining an initial set of trajectories through a search on a graph describing the connectivity of the cells' ports. The restriction of the motion to only 3 primitives, allows an agent to access one of exactly three adjacent cells every time. Thus a directed graph modeling the connectivity of the cells can be generated in a systematic way.

We shortly remind that, according to graph theory, a graph, $\mathcal{G}$ is a set of vertices $\mathcal{V}$ and an associated set of unordered pairs of vertices, $\mathcal{E}$, called *edges*. Two vertices, $v_i, v_j \in \mathcal{V}$, are said to be *connected* iff $(v_i, v_j) \in \mathcal{E}$. The edges of a graph may have real numbers w_i called *weights* associated with them. A directed graph is a graph whose set of edges, $\mathcal{E}$ comprises *ordered* pairs of vertices.

We can now construct the graph denoting the connectivity of the workspace cells. The vertices of the graph are the tuples $v = (i, j, d)$, with (i, j) valid workspace cells and $d \in \mathcal{D}$. For each vertex v_i, the edges $(v_i, g(v_i, m)), m \in \{-1, 0, 1\}$ are created.

A set of trivial trajectories, $\{\mathcal{P}_i\}$, is obtained by performing an ordinary graph search, using e.g. A* or Dijkstra's algorithm for each agent independently. This set of trajectories is not necessarily collision-free. Subsequently, the following algorithm tries to incrementally resolve all conflicts, within

a short time-window. A short description of the algorithm follows.

```
1  while true:
2    if not collision(agents.trajectories):
3      break
4    else:
5      segments := []
6      collision := first_collision(agents.trajectories)
7      if livelock():
8          break
9      collision_point := collision_point(collision)
10     agents := get_agents(collision)
11     for agent in agents:
12       segment :=
13         segment(agent.trajectory, collision_point, window)
14       segments.append(segment)
15     new_segments := deconflict(segments)
16     for agent in agents:
17       agent.trajectory := replace(segment, new_segment)
```

- *collision()* checks whether there is a collision among the agents' trajectories, as per section VI.
- *first_collision()* returns the temporally first collision among all collisions.
- *collision_point()* returns the cell in which the collision takes place, in the grid coordinate system (i, j), as per section IV and *get_agents()* returns the set of all agents participating in the collision.
- *segment()* returns a segment of size *window*, centered around the collision point. If the window contains the start or the end of a trajectory, the segment is truncated accordingly.
- *deconflict()* attempts to resolve the collision within the specified window, using the model-checking approach described above. In this context, the segments of the colliding agents that fit in the window are treated as a new system, whose specification is fed to the model checker. The model-checker verifies the specification and searches for a counterexample as proposed in section VII. Subsequently, *replace()* replaces the conflicting segments with their collision-free replacements calculated by the model checker in the previous step.
- *livelock()* detects whether the algorithm is engaged in a livelock. Since the resolution of a collision may create another collision point, it is possible that a livelock will emerge. Livelocks can be detected by remembering which collision points have been handled so far and which agents were participating and checking for duplicates.

The process runs iteratively, resolving collisions that appear first in the problem, until there are no more collisions left to resolve. However, as was pointed out earlier, the use of a model checker as a local solver implies a trade-off: The algorithm tries to solve the problem in a limited space, where a solution may not exist. If a solution does not exist, then the problem has to be solved anew in the whole space.

Simulation results are discussed in the following section.

IX. SIMULATION RESULTS

Fig. 7(d) depicts the solution given to a three-agent problem by feeding the complete model of the system to the model checker. Figs. 7(a-c) give detailed, snapshots of the resulting trajectories near what would otherwise have been

(a) $t = 3$ (b) $t = 5$ (c) Goals reached
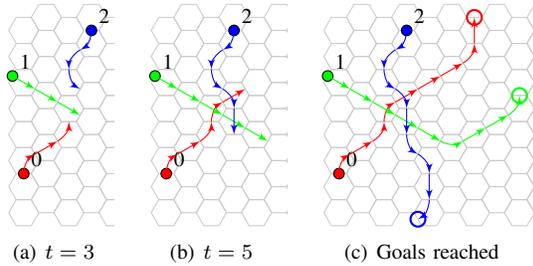
Fig. 7. Snapshots of the solution in a three-agent scenario. The filled circles represent the start positions, circle outlines mark the goal positions.

(a)   (b)   (c)   (d)

Fig. 8. A run of the incremental algorithm proposed in Section VIII. The filled circles mark the agents' initial positions and the circle outlines mark their respective goal positions. See Section IX for a detailed discussion.

a double collision point. This solution is the result of a full state-space search performed by the model checker on the complete 3-agent system. Every snapshot shows the initial position of the agents (opaque circles), the trajectory traveled so far and the current position (last arrow tip). Notice that both collision cases, as outlined in section VI are avoided.

Fig. 8 shows snapshots from the run of the hierarchical algorithm described in section VIII. The initial set of trajectories was obtained by a shortest-path search on the graph constructed according to section VIII, using Dijkstra's algorithm is shown in 8(a). The agent start positions are marked using circles, labeled with a corresponding identification number. Note that the graph search does not make any collision checks. The initial set of trajectories contains collisions between agents 0 and 1, 1 and 2, and 2 and 3.

The collision points are identified using the algorithm provided in section VI and the temporally first collision, the one between agents 1 and 2 is considered first. A model checking specification is generated to replace the path segments of 5 time-steps around the collision point. The collision points of the new trajectory set are re-calculated and the temporally first collision point is chosen, this time between agents 0 and 1. Both collisions are resolved in Fig. 8(b) and Figs. 8(c) - 8(d) show the resolution of the final conflict between agents 2 and 3. The initial and final positions of the agents were chosen so that the agent paths in cases (0,1) and (2,3) coincided and resulted in a heads-on collision, in order to demonstrate the ability of the algorithm to handle symmetry and new collision points that emerge as a side-effect of previous de-confliction operations. The simulation was based on custom-written Python software performing the graph search and system specification generation and NuSMV as a model checker backend. The runtime was 20 seconds on a 1.8GHz Core2Duo CPU. For comparison, using NuSMV on the whole problem did not yield any results within 5 hours.

X. CONCLUSIONS AND FUTURE RESEARCH

In this paper we demonstrated how a problem with multiple non-holonomic agents can be reduced - through appropriate abstraction - to a form solvable using formal verification tools. Furthermore we introduced a way to use this approach to refine a trivial solution. The algorithms proposed were backed by non-trivial simulation data.
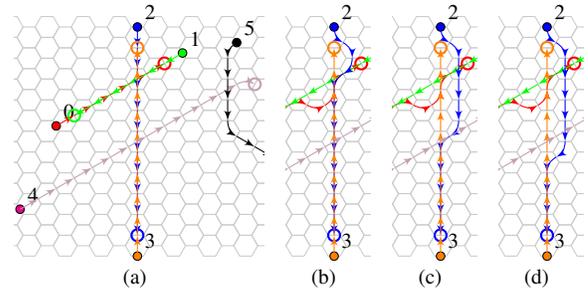
Future research directions include the extension to a distributed version, new ways to reduce the computational complexity of the system, ways to incorporate "user" requirements as to the trajectories generated and the incorporation of complex stationary obstacles and non-cooperating agents.

REFERENCES

[1] E. Rimon and D. E. Koditschek, "Exact robot navigation using artificial potential fields," *IEEE Transactions on Robotics and Automation*, vol. 8, no. 5, pp. 501–518, October 1992.
[2] S. G. Loizou and K. J. Kyriakopoulos, "A feedback based multiagent navigation framework," *International Journal of Systems Science*, vol. 37 (6), pp. 377–384, 2006.
[3] ——, "Navigation of multiple kinematically constrained robots," *IEEE Transactions on Robotics*, vol. 24 (1), pp. 221–231, 2008.
[4] C. Tomlin, G. J. Pappas, and S. Sastry, "Conflict resolution for air traffic management: A study in multiagent hybrid systems," *IEEE Transactions on Automatic Control*, vol. 43, no. 4, 1998.
[5] S. G. Loizou and K. J. Kyriakopoulos, "Automatic synthesis of multi-agent motion tasks based on ltl specifications," in *Proceedings of the 43rd IEEE Conference on Decision and Control*. IEEE, 2004.
[6] G. E. Fainekos, H. Kress-Gazit, and G. J. Pappas, "Temporal logic motion planning for mobile robots," in *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, April 2005, pp. 2020–2025.
[7] H. Kress-Gazit, G. Fainekos, and G. J. Pappas, "From structured english to robot motion," in *Proceedings of the 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2007.
[8] M. M. Quottrup, T. Bak, and R. Izadi-Zamanabadi, "Multi-robot planning: A timed automata approach," in *2004 IEEE International Conference on Robotics and Automation*, 2004.
[9] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems," in *Proc. of Workshop on Verification and Control of Hybrid Systems III*, ser. Lecture Notes in Computer Science, no. 1066. Springer–Verlag, Oct. 1995, pp. 232–243.
[10] C. Belta, A. Bicchi, M. Egerstedt, E. Frazzoli, E. Klavins, and G. J. Pappas, "Symbolic planning and control of robot motion," *IEEE Robotics and Automation Magazine - special issue on Grand Challenges of Robotics*, vol. 14, no. 1, pp. 61–71, 2007.
[11] G. J. Pappas, "Hybrid systems: Computation and abstraction," Ph.D. dissertation, Univ. California at Berkeley, 1998.
[12] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994.
[13] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking," in *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, ser. LNCS, vol. 2404. Copenhagen, Denmark: Springer, July 2002.