DRONA: A Framework for Safe Distributed Mobile Robotics

Ankush Desai$^1$, Indranil Saha$^2$, Jianqiao Yang$^1$, Shaz Qadeer$^3$, Sanjit A. Seshia$^1$

$^1$UC Berkeley $^2$IIT Kanpur $^3$Microsoft Research

{ankush,jq.yang,sseshia}@eecs.berkeley.edu, isaha@cse.iitk.ac.in, qadeer@microsoft.com

ABSTRACT

Distributed mobile robotics (DMR) involves teams of robots navigating in a physical space to achieve tasks in a coordinated fashion. A major challenge in DMR is to program the ensemble of robots with formal guarantees and high assurance of correct operation. To this end, we introduce Drona, a framework for building reliable DMR applications.

This paper makes three central contributions: (1) We present a novel and provably correct decentralized asynchronous motion planner that can perform on-the-fly collision-free planning for dynamically generated tasks. Moreover, the motion planner is the first to take into account the fact that distributed robots may have clocks that are only synchronized up to a tolerance, i.e., they are almost synchronous; (2) We formalize the DMR system as a mixed-synchronous system, and present a sound abstraction-based verification approach for DMR systems, and (3) Drona provides a state-machine based language for safe event-driven programming of a DMR system and the generated C code by the compiler can be executed on robot operating system (ROS).

To demonstrate the efficacy of Drona, we build and verify an example priority mail delivery system. Using our abstraction-based verification approach we were able to find, within a few minutes, bugs which could not be found by performing random simulation for several hours. Our verified decentralized motion-planner scales efficiently for large number of robots (upto 128 robots) and workspace sizes (upto a 256x256 grid).

1. INTRODUCTION

Recent demonstrations of autonomous robots collaborating to accomplish complex missions have fueled excitement about the future opportunities offered by distributed mobile robotics (DMR). Applications of DMR systems span a broad spectrum of areas like surveillance, law enforcement, agriculture, disaster management, warehouse and delivery systems. As DMR systems are becoming increasingly prevalent, programmability with high assurance and provable guarantees is a major barrier to their large scale adaptation.

In this paper, we consider a class of DMR systems where a fixed set of robots shares a known workspace with static obstacles and the tasks to be performed by the system are generated dynamically. Safe programming of such a DMR system is notoriously hard as the programmer has to correctly reason about failures, uncertain environments, asynchrony, dynamically generated tasks and interfering robots in the workspace. To address this problem, we present Drona, a software framework that helps build reliable DMR systems.

Example DMR application: Fig. 1 shows the 2D representation of a city area in which a fleet of drones operates to pickup and deliver packages. The black blocks represent buildings and are the static obstacles in the workspace. The dotted blocks are battery charging locations that the drones must visit to charge their batteries. Mail delivery tasks are nondeterministically generated by the environment with associated mail priority, the package pickup location and the drop off (goal) location. We use the mail delivery system as an example DMR application in rest of the paper and demonstrate how Drona can be used for the safe programming of such applications.

When a team of mobile robots shares the same workspace, one of the fundamental problems is to prevent collisions and still compute optimal paths for the
individual team members. For example, in the *mail delivery system*, mail requests are generated in real-time and the drones might have to move simultaneously in the workspace computing collision-free paths on-the-fly. To address this problem, **Drona** implements a provably correct multi-robot motion planner (MRMP) which is decentralized, asynchronous, and reactive to dynamically generated task requests.

Prior work on multi-robot motion planning (e.g. 28, 29, 27, 24, 23) makes an assumption that the robots in the system step synchronously, or in other words, their local clocks are synchronized. In distributed systems there is no perfect synchrony and this unsound assumption can lead to motion planner computing colliding trajectories. With the advances in time-synchronization protocols [10], clocks in the distributed system can be synchronized within a small bound. One of the salient feature of MRMP is that it does not assume perfect synchrony of the distributed clocks. It produces safe collision-free trajectories taking into account the “almost synchrony” in a time-synchronized DMR system.

We take a principled approach towards implementing a generic DMR software stack in **Drona** with MRMP as its core component (Sec. 2.1). A DMR system implemented using **Drona** consists of both event-driven asynchronous processes and periodic processes. For verification of such a DMR system, we formalize it as a mixed-synchronous system and implement an abstraction-based model-checking approach in **Drona**.

A major challenge in programming autonomous robots is to correctly handle nondeterministically generated events and their interleavings. We integrate a state-machine based programming language **P** [7] into the **Drona** tool-chain. **P** simplifies the process of implementing and specifying event-driven asynchronous programs. The generated C code from the high-level **P** program can be directly deployed on Robot Operating System (ROS) 23. **P** supports **Zing** 3, 8, a state-of-the-art model-checker for verification of **P** programs. We extended **Zing** to implement the mixed-synchronous abstraction based model-checking approach for verification of DMR system.

To demonstrate the efficacy of the **Drona** tool-chain we implemented and verified the *mail delivery* system. Using the abstraction-based verification approach we found several critical bugs in our implementation of the application and software stack which the random simulation based approach failed to find. Our results show that MRMP scales efficiently for systems with large number of robots (upto 128 robots), and hence can be used for on-the-fly computation of safe-trajectories in real systems. **Drona** tool-chain and simulation videos of some of our experiments are publicly available 2.

We summarize the contributions of our paper:

1. We present a novel and provably correct decentralized asynchronous motion planner that can perform on-the-fly collision-free planning for dynamically generated tasks. Moreover, the motion planner is the first to take into account the fact that distributed robots may have clocks that are only synchronized up to a tolerance.
2. We formalize the DMR system as a mixed synchronous system and implement a sound abstraction-based model checking approach in **Drona** for verifying DMR systems.
3. We demonstrate the advantages of using **Drona** for safe programming and verification of DMR systems by implementing the *mail delivery system* as a case study. The generate C code from our framework can be easily deployed on ROS simulator.

## 2. PRELIMINARIES

In this section, we first provide an overview of the DMR software stack implemented in **Drona**, followed by the set of definitions used in the rest of the paper.

### 2.1 Overview of DMR Software Stack

Fig. 2 presents the modular software stack executed by each robot in the system built using **Drona**. The edges in Fig. 2 represent interaction between modules, these modules interact by sending events asynchronously.

At the top is the task-planner (TP) that implements the application specific protocol to guarantee that the system satisfies application-specific goals. For example, the task-planner for mail delivery system is responsible for ensuring that the mail requests are handled responsively and are always delivered in priority order. Whenever task-planner wants the robot to perform a task by going to a location it sends a request to the motion-planner to compute a trajectory to the goal location. It is the role of multi-robot motion planner (MRMP) module to compute safe and collision-free trajectory for the robot by coordinating with other robots in the system. On computing a trajectory, the motion-planner sends the trajectory to the plan-executor module.

![Figure 2: Software Stack](image)

The plan-executor (PE) module guarantees that the robot correctly follows the trajectory computed by the motion planner. The sense and infer (SI) module implements the monitoring state-machines that continuously monitor the sensor streams coming from the robot and informs the task-planner only if it infers an event that requires task planner’s attention. For example, the BatteryMonitorMachine monitors the battery sensor data-stream coming from the robot and informs the task-planner when the battery level is less than a...
threshold. The task-planner then responds to the critical event and takes the drone to the battery charging point. Robot manufacturing companies also provide a software development kit (SDK) that implements basic primitives for programatically controlling a robot, sampling its state and communicating with other robots in the system. More details about the software stack provided by Drona is available at [2].

2.2 Terminology and Definitions

In this section, we formalize the definitions needed for the rest of the paper.

**Workspace:** We represent the workspace for a DMR application as a 3-D occupancy grid map, the top view of an example 3-D workspace is shown in Fig. 1. The grid decomposes the workspace into cube-shaped blocks. The size of a workspace is represented using the number of blocks along each dimension. For example, if the workspace contains \(n_x\), \(n_y\) and \(n_z\) blocks along the x, y and z dimension respectively, the size of the workspace is represented as \([n_x \times n_y \times n_z]\). Each block is assigned a unique identifier (Fig. 1) which represents the location of that block in the workspace. The set of all locations in the workspace is denoted by the set \(W\). Some parts of the workspace can be occupied by static obstacles. If a grid block is partially occupied by an obstacle, we mark the entire grid block to be covered by obstacle. The set of locations covered by obstacles is denoted by \(\Omega\). The fixed set of robots operating in the workspace is denoted by the set \(\Gamma\). For a grid location \(l\), an obstacle \(\gamma\) at location \(l\) (including \(l\) and \(\text{post}(l, \gamma)\)). For a motion primitive \(\gamma \in \Gamma\), we denote by \(\text{cost}(\gamma)\) the cost (e.g., energy expenditure) to execute the motion primitive. We assume that for all robots in the system, each motion primitive requires \(\tau\) unit time for execution. This assumption may not hold for heterogeneous systems and extending our approach for such systems is left as a future work.

**Motion plan:** Now we formally define a motion plan.

**Definition 1 (Motion Plan).** A motion plan is defined as a sequence of motion primitives to be applied to a robot \(r_i\) to move from an initial location \(l_i^0\) to a goal location \(l_i^\gamma\). A motion plan is denoted by \(\rho_i = (\gamma_1 \ldots \gamma_k)\), where, \(\gamma_q \in \Gamma\) for \(q \in \{1, \ldots, k\}\).

**Timed trajectories:** The continuous motion of robot \(r_i\) in the system can be represented as a sequence of timestamped locations \((l_i^n, l_i^\gamma)\), \((l_i^n, l_i^\gamma)\) \ldots where \(l_i^n\) represents the \(n\)-th periodic time step for robot \(r_i\). In the rest of the paper we refer to \((l_i^n, l_i^\gamma)\) as \(l_i^n\) representing the location of robot \(r_i\) in the \(n\)-th time step. The size of the period \(|l_i^n - l_i^{n+1}| = \tau\), where \(\tau\) is the time it takes to execute any motion primitive.

**Definition 2 (Trajectory).** Given the current location \(l_i^0\) of the robot \(r_i\) and a motion plan \(\rho_i = (\gamma_1 \ldots \gamma_k)\) that is applied to the robot at the time step \(l_i^n\), the trajectory of the robot is a sequence of locations \(\xi_i = (l_i^n, l_i^\gamma, \ldots, l_i^{n+k})\), such that \(l_i^n = l_i^\gamma\), for all \(q \in \{0, \ldots, k-1\}\), \(\gamma_{q+1}\) is applied to the robot at location \(l_i^{n+q}\) at the time step \(l_i^{n+q}\) and \(l_i^{n+q+1} = \text{post}(l_i^{n+q+1}, \gamma_{q+1})\).

**Safe-trajectory property:** The trajectory computed by the motion planner must always satisfy the safe-trajectory property \(\Phi_{st}\) which is a conjunction of following three properties: (a) obstacle avoidance \(\phi_o\), (b) collision avoidance \(\phi_c\), and (c) successful task completion \(\phi_f\). The property \(\phi_c\) requires that a robot never attempts to pass through a location \(l \in \Omega\) associated with a static obstacle. The property \(\phi_c\) entails that two robots never collide with each other. The property \(\phi_f\) captures the requirement that if the robot follows the trajectory then it will eventually reach the goal location.

3. MULTI-ROBOT MOTION PLANNER

In this section, we present the multi-robot motion planner (MRMP) implemented in Drona. MRMP is asynchronous, decentralized, and robust to clock skew in distributed systems.

The motion planning problem in DMR can be described as follows:

**Problem 1.** Given a set of robots \(R = \{r_1, \ldots, r_{|R|}\}\) operating in a common workspace \(W\), if a dynamically generated test \((l, p)\) \(\in T\) is assigned to a robot \(r_i \in R\), find trajectory \(\xi_i\) such that it satisfies safe-trajectory property \(\Phi_{st}\).
We decompose the above motion planning problem into two sub-problems:

1. **Trajectory coordination problem**: For computing the collision-free trajectory of a robot, motion planner must have a consistent snapshot of the trajectories of all other robots in the system (Sec. 3.1).

2. **Safe plan-generation problem**: Given the set of current trajectories of all the robots ($\Psi$), synthesize a safe trajectory that is robust against time-synchronization errors in distributed systems and satisfies $\Psi_{st}$ (Sec. 3.2).

### 3.1 Distributed Asynchronous Coordination

In a DMR system, tasks are generated dynamically. Hence, the motion planner for such a system should be able to compute safe trajectories on-the-fly and in a decentralized fashion.

The decentralized motion-planner for robot $r_i \in R$ is shown in Protocol 1 in the form of a state machine, which is executed by each robot in the system. It is presented in the form of pseudo-code that closely represents the syntax of P programming language for state machine based systems. A P program comprises of concurrently executing state machines (a.k.a. actors) communicating asynchronously with each other using events accompanied by typed data values. Each state machine has an input queue and machine-local store for a collection of variables. On entering a state, the entry function corresponding to that state is executed. Each state has a set of event-handlers which get executed on receiving the corresponding event. The function send $(tr, ev, pd)$ is used to send a event $ev$ with payload data $pd$ to target machine $tr$. The function broadcast $(ev, pd)$ broadcasts event $ev$ with payload $pd$ to all the robots in workspace, including oneself.

The motion-planner state machine has three states: WaitForTaskRequest, CoordinateAndGeneratePlan, and WaitForPlanCompletion. Planner starts executing in the WaitForTaskRequest state. On receiving a NewTask event from the task-planner, it updates the task information (currTaskId and $l_i^c$) and moves to the CoordinateAndGeneratePlan state. If the planner receives a ReqForCurrentTraj event from another robot $r_j \in R$, it sends its current location $l_i^c$ to $r_j$.

Upon entering the CoordinateAndGeneratePlan state, planner broadcasts ReqForCurrentTraj event with the identifier of the current task and its own identifier, asking for trajectories of all robots in the workspace. $R_{recv}$ set stores identifiers of the robots that have sent their trajectories as a response to the ReqForCurrentTraj event, and $\Psi_i$ set contains the current trajectories of all those robots. $R_{pend}$ is used for storing identifiers of all robots from which it has received ReqForCurrentTraj and have to send its newly computed trajectory.

Upon receiving the CurrentTraj event from another robot $r_j$, the planner adds robot $r_j$ to set $R_{recv}$ and its trajectory $\zeta_j$ to the set $\Psi_i$. The planner state machine is blocked in CoordinateAndGeneratePlan state until it receives CurrentTraj event from all the robots.

### Protocol 1 Decentralized Motion Planner

1: machine DECENTRALIZED MOTION PLANNER
2: start state WAIT FOR TASK REQUEST
3: entry { $l_i^c \rightarrow$getCurrentLocation() }
4: on ReqTask (task : T) do {
5: currTaskId $\leftarrow$ task.id, $l_i^c \leftarrow$ task.goal
6: goto COORDINATE AND GENERATE PLAN
7: }
8: on ReqForCurrentTraj (taskId, $r_j$) do {
9: send ($r_j$, CurrentTraj, ($r_i$, $l_i^c$))
10: }
11: }
12: state COORDINATE AND GENERATE PLAN
13: entry {
14: $R_{pend}$ $\leftarrow$ {}, $R_{recv}$ $\leftarrow$ {}, $\Psi_i$ $\leftarrow$ {}
15: broadcast (ReqForCurrentTraj, (currTaskId, $r_i$))
16: }
17: on CurrentTraj ($r_j$, $\zeta_j$) do {
18: $R_{recv}$ $\leftarrow$ $R_{recv}$ $\cup$ {$r_j$}, $\Psi_i$ $\leftarrow$ $\Psi_i$ $\cup$ {$\zeta_j$}
19: if (sizeof($R_{recv}$) = |$R$|) then
20: $\rho_i$ $\leftarrow$ synthesizeMotionPlan($l_i^c$, $l_j^c$, $\Omega$, $\Psi_i$)
21: SendMotionPlanToPlanExecutor($\rho_i$
22: $\xi_i \leftarrow$ ConvertMotionPlanToTraj($\rho_i$)
23: foreach $r_j \in R_{pend}$
24: send ($r_j$, CurrentTraj, ($r_i$, $\xi_i$))
25: }
26: goto WAIT FOR PLAN COMPLETION
27: }
28: on ReqForCurrentTraj (taskId, $r_j$) do {
29: if (taskId $\leq$ currTaskId) then
30: send ($r_j$, CurrentTraj, ($r_i$, $l_i^c$))
31: else $R_{pend}$ $\leftarrow$ $R_{pend}$ $\cup$ {$r_j$}
32: }
33: end if
34: }
35: }
36: }
37: state WAIT FOR PLAN COMPLETION
38: on ReqForCurrentTraj (taskId, $r_j$) do {
39: send ($r_j$, CurrentTraj, ($r_i$, $\zeta_j$))
40: }
41: on Reset () do {
42: goto WAIT FOR TASK REQUEST
43: }
44: }
45: }

On receiving trajectories from all the robots (line 19), the planner invokes the synthesizeMotionPlan function with its current location $l_i^c$, the goal location $l_j^c$, the set of static obstacles $\Omega$ and the set of trajectories of all the robots $\Psi_i$. The implementation of plan generator function synthesizeMotionPlan is described in Sec. 3.2. The motion-plan returned by the synthesizeMotionPlan function is sent to the plan-executor module so that the robot can start executing it, and the corresponding trajectory is sent to all the robots whose identifiers are present in the set $R_{pend}$ and are blocked waiting for the trajectory of robot $r_i$.

If two robots $r_i$ and $r_j$ attempt to generate motion plans for two tasks simultaneously then a race situation arises as both of them are waiting for the current trajectory of the other robot. This deadlock situation is resolved based on the unique identifier assigned to the tasks. If the planner of $r_i$ receives a ReqForCurrentTraj event from $r_j$ in the CoordinateAndGeneratePlan state and task identifier taskId in the event is less than...
its current task identifier \textit{currTask}_{i} indicates that the robot \( r_j \) is dealing with a higher priority task, the motion planner of \( r_i \) sends its current location \( l^i \) to the motion planner of \( r_j \) to unblock it and waits for \( r_j \)'s computed trajectory. Otherwise, it adds the robot \( r_j \) to the set \( R_{\text{pend}} \), and once it computes its own trajectory, sends the trajectory to unblock \( r_j \) (Line 23-25).

In the \textit{WaitForPlanCompletion} state, motion planner waits for a \texttt{Reset} event from the plan-executor indicating that the task is completed, on receiving which it moves to \textit{WaitForTaskRequest}.

Notice that if the planner for robot \( r_i \) generates trajectory \( \xi_i \), then \( \xi_i \) is always safe as the coordination protocol guarantees that all future trajectories computed by any other robot \( r_j \) will have \( \xi_i \) in \( \Psi_j \).

### 3.2 Safe Plan Generator

In this section, we present an approach for synthesizing a motion plan to generate a trajectory that satisfies the safe-trajectory property \( \Phi_{st} \).

#### 3.2.1 Motion Plan Synthesis Problem

The inputs to the motion plan synthesis problem (Protocol 1 line 20) for a robot \( r_i \) is the current location of the robot \( (l^i) \), the goal location \( (l_g) \), the set of static obstacles \( (\Omega) \), and the set of current trajectories of other robots \( (\Psi_j) \). We call the tuple \( \mathcal{P}_i = (l^i, l_g, \Omega, \Psi_j) \) as the \textit{motion plan synthesis problem} instance for robot \( r_i \).

Recall that a trajectory \( \xi_i \) of robot \( r_i \) is a sequence of locations \( (l^i_n, l^i_{n+1}, \ldots, l^i_{n+k}) \), where the trajectory starts at the \( n \)-th time step. We adopt a technique based on composition of motion primitives [24][25] to solve the motion-plan synthesis problem. To generate such a trajectory \( \xi_i \), we must synthesize a motion plan \( \rho_i = (\gamma_1, \gamma_2, \ldots, \gamma_k) \), where \( \gamma_q \in \Gamma \), \( 1 \leq q \leq k \) (Def. 1). To follow the desired trajectory, the robot must start executing the first motion primitive precisely at the beginning of the \( n \)-th period, i.e., at time \( \tau^i_n \) according to its local clock.

We now define the motion plan synthesis problem:

**Problem 2.** Given a motion plan synthesis problem instance \( \mathcal{P}_i \) for robot \( r_i \), a set of motion primitives \( \Gamma \), and the time step \( \tau^i_n \) when the plan executor will start executing the motion plan, synthesize a motion plan \( \rho_i = (\gamma_1, \gamma_2, \ldots, \gamma_k) \) such that the trajectory \( \xi_i = (l^i_n, l^i_{n+1}, \ldots, l^i_{n+k}) \) generated by the plan executor by executing the motion plan \( \rho_i \) satisfies the safe-trajectory property \( \Phi_{st} \).

**Accounting for clock skew:** Each robot \( r_i \in R \) operates based on its own local clock \( \chi_i \). Let \( t \) denote an ideal global time reference (just for purposes of formalization). We denote by \( \chi_i(t) \) the evaluation of the clock at the global time \( t \). Synchronization of these clocks plays an important role in the correctness of our distributed motion planning algorithm with respect to the collision avoidance property \( \phi_c \). We assume that the DMR software stack implements a time-synchronization protocol [16] that bounds the skew between two clocks, given by \( |\chi_i(t) - \chi_j(t)| \leq \beta \). If \( \beta = 0 \), we say that the clocks of the robots are in \textit{perfect synchrony}. In general, however, \( \beta > 0 \), and we define this more general condition below.

To capture the skew between timed trajectories of two robots, we define a parameter \( \Delta \) that denotes the maximum offset between the sequences of steps \( \tau^i \) and \( \tau^j \) of any two robots \( r_i \) and \( r_j \).

**Theorem 1.** If the local clocks of robots \( r_i \) and \( r_j \) are \textit{time-synchronized} with a synchronization precision \( \beta \), and at some global time point \( t \), if robot \( r_i \) takes the time step \( \tau^i_p \) and robot \( r_j \) takes the time step \( \tau^j_q \), then \( |p-q| \leq \Delta \), where \( \Delta \) is given by \( \Delta = \lceil \frac{\beta}{\tau} \rceil \) where \( \tau \) is the duration of a time step.

The above condition is called \textit{approximate synchrony} and was introduced and proved in our previous work [9]. Informally, Theorem 1 states that if the clocks of two robots are synchronized within a bound \( \beta \) then the difference between the number of periodic steps taken by the two robots is bounded by \( \Delta \).

Hence, for collision avoidance, while synthesizing motion plan it is important to know precisely where the other robots in the system would be for a time-step window of size \( \pm \Delta \). The parameter \( \Delta \) determines how conservative a robot should be, when computing its trajectory that avoids collision with other robots.

#### 3.2.2 Motion Plan Generation

We now describe how a motion plan \( \rho_i = (\gamma_1, \ldots, \gamma_k) \) for a robot \( r_i \) is synthesized from a motion plan synthesis problem instance \( \mathcal{P}_i = (l^i, l_g, \Omega, \Psi_j) \).

We formulate the problem as an optimization problem where the decision variables are the motion primitives to be applied at different time steps, and the objective is to minimize the total cost to execute the trajectory. The functions \texttt{post}, \texttt{cost}, and \texttt{intermediate} used in this section are defined in Sec. 2.2.

The objective function is given as follows:

\[
\text{minimize } \sum_{j=1}^{k} \text{cost}(\gamma_j)
\]

(1)

The constraints for the optimization problem is a conjunction of four constraints as described below:

**1) Initial and final location:** The first location in \( \xi_i \) is the current location \( l^i_c \) of the robot. Similarly, the last location in \( \xi_i \) matches with the goal location \( l_g \).

\[
l^i_n = l^i_c \land l^i_{n+k} = l_g
\]

(2)

**2) Trajectory continuity:** A location in a trajectory is reachable from the previous location using the motion primitive applied to the previous location.

\[
\forall q \in {0, \ldots, k-1} : l^i_{n+q+1} \in \text{post}(l^i_{n+q+1}, \gamma_{q+1})
\]

(3)
(3) **Obstacle avoidance:** No location on the trajectory should be covered with obstacles.

\[ \forall q \in \{0, \ldots, k-1\} \forall l \in \text{intermediate}(t_{n+q},\tau_{n+1}) : l \notin \Omega \quad (4) \]

This constraint ensures the obstacle avoidance component \( \phi_o \) of the safe-trajectory property \( \Phi_{st} \).

(4) **Collision avoidance:** If the local clocks of all the robots are in perfect synchrony, ensuring collision avoidance would require that the robots do not occupy the same grid location in the workspace at the same time period according to their local clock. Motion plan synthesizer must ensure collision avoidance of robot \( r_i \)'s trajectory represented as \( \xi_i = (l_{n_i}^{t_n}, l_{n+1}^{t_n}, \ldots, l_{n+r_i}^{t_n}) \) with the trajectories of other robots captured in the set \( \Psi \). The trajectory of any other robot \( r_j \) is denoted by \( (l_{n_j}^{t_n}, l_{n+1}^{t_n}, \ldots, l_{n+m_j}^{t_n}) \) \( \in \Psi \), where \( m < n \).

The following constraint guarantees collision avoidance property \( \phi_c \) for a perfectly synchronous system:

\[ \forall r_j \in R \setminus \{ r_i \}, (l_{n_j}^{t_n}, l_{n+1}^{t_n}, \ldots, l_{n+m_j}^{t_n}) \in \Psi : \]

\[
(\langle \forall q \in \{n, \ldots, \min(n', m')\} \forall p \in \{q - \Delta, \ldots, q + \Delta\} : \\
(n \leq p \leq m' \Rightarrow l_p^{t_n} \neq l_p^{t_m}) \land \\
(p < m' \Rightarrow q \in \{n' + 1, \ldots, m'\} : l_p^{t_n} \neq l_p^{t_{n'}}) \land \\
(p > m' \Rightarrow q \in \{m' + 1, \ldots, n\} : l_p^{t_n} \neq l_p^{t_{m'}})) \quad (5)
\]

Once a robot reaches its destination, it stays there unless it computes a new trajectory using the motion planner. Eq. (5) comprises conjunction of three constraints (one per line). The first constraint enforces that two robots cannot occupy the same location at the same instant while moving. The second and third constraint specify that a robot that is moving does not occupy the location of a stationary robot (that has stopped after reaching destination).

When the clocks are not perfectly synchronous, then one must consider the synchronization precision \( \beta \). We do so using the notion of approximate synchrony introduced in Theorem 1. Specifically, to ensure collision avoidance with another robot, the plan synthesizer of a robot should ensure that its location at time step \( r_n \) does not overlap with the location of the other robot at any step in the range of \( (r_n^{\tau_n} - \Delta, r_n^{\tau_n} + \Delta) \).

Eq. (6) extends Eq. (3) to encode collision avoidance constraint with an approximate synchrony bound of \( \Delta \).

\[ \forall r_j \in R \setminus \{ r_i \}, (l_{n_j}^{t_n}, l_{n+1}^{t_n}, \ldots, l_{n+m_j}^{t_n}) \in \Psi : \]

\[
(\langle \forall q \in \{n, \ldots, \min(n', m')\} \forall p \in \{q - \Delta, \ldots, q + \Delta\} : \\
(n \leq p \leq m' \Rightarrow l_p^{t_n} \neq l_p^{t_m}) \land \\
(p < m' \Rightarrow q \in \{n' + 1, \ldots, m'\} : l_p^{t_n} \neq l_p^{t_{n'}}) \land \\
(p > m' \Rightarrow q \in \{m' + 1, \ldots, n\} : l_p^{t_n} \neq l_p^{t_{m'}})) \quad (6)
\]

The above constraints can be directly encoded in the input language of an SMT solver and the solution obtained from the solver can be used to generate a motion plan for the robot \( r_i \). However, as our experimental results reveal (Sec. 6), SMT solver based solution suffers from lack of scalability for large grid sizes and multi-robot systems as constraints become hard to solve.

**A* based safe plan-generator:** To have a scalable implementation, we extend the well-known A* search algorithm [13] to generate safe motion plans. A* search algorithm can natively handle the objective function Eq. (1) and the constraints Eq. (2)-(4) for static obstacles. We extended the function that computes adjacent nodes in A* to incorporate the constraints in Eq. (5) and Eq. (6). We associate a time-stamp value to each node in the A* search graph. The time-stamp denotes the number of steps required to reach the current node from the start node. During adjacent node calculation, we use time-stamp at a node to check the constraints in Eq. (5) and Eq. (6) to ensure that the trajectory through the potential adjacent node will not be in collision with the trajectory of any other robot.

### 3.3 Plan Executor

The plan-executor (PE) module plays an important role in the overall correctness of MRMP. It is the responsibility of the plan-executor module to ensure that the robot correctly follows its computed trajectory. The plan-generator (Sec. 3.2) generates safe trajectory under the assumption that all robots in the system will follow their timed-trajectories that they communicated to other robots. Recollect that the MRMP protocol (Protocol 1 line 21) on computing a motion plan \( \rho_i \) sends it to the plan-executor module. The plan-executor executes the sequence of motion-primitives in \( \rho_i \) such that the robot \( r_i \) realizes its timed-trajectory \( \xi_i \) (Def. 2). It is implemented as a periodic state-machine with the duration of each period as \( \tau \), executing the next motion-primitive at each period.

For all the robots to follow their timed-trajectories correctly, the path-executor processes across robots must step periodically with a symmetric period \( \tau \), i.e., \( \forall r_i \in R, \forall n, |r_n^{\tau_n} - r_{n+1}^{\tau_{n+1}}| = \tau \). Since path-executor at each robot \( r_i \) step using its local clock \( \chi_i \), the path-executors across the system do not step perfectly synchronously but almost-synchronously with a bound \( \pm \Delta \) which the plan-generator has accounted for in Eq. (6).

### 3.4 Provably Correct Motion Planner

We perform assume-guarantee style reasoning for proving the correctness of MRMP. Recollect that when computing a trajectory for a robot \( r_i \), the execution of MRMP can be decomposed into two phases: first the coordination protocol computes the avoid trajectories set \( \Psi_i \), which is then used by the safe plan-generator for computing the collision-free trajectory \( \xi_i \). We say that the avoid trajectories set \( \Psi_i \) is consistent if \( \forall \xi_j \in \Psi_i, \xi_j = \xi_j \), where \( \xi_j \) is the trajectory sent by
robot \(r_j\) to robot \(r_i\) and \(\xi_j\) is the actual trajectory being executed by robot \(r_j\).

The \(A^*\) based plan-generator described in Sec. 3.2.2 always generates trajectories that satisfy the safe-trajectory property \(\Phi_{st}\) under the assumption that avoid trajectory set \(\Psi_i\) is consistent. In other words, given the set of trajectories \(\Psi_i\), if the plan-generator computes trajectory \(\xi_i\) then consistent(\(\Psi_i\)) \(\implies (\xi_i \models \Phi_{st})\).

In order to prove that the assumption consistent(\(\Psi_i\)) holds, we verify (using model-checking) the following properties about the coordination protocol: (1) Safety: The avoid trajectory set \(\Psi_i\) computed by the coordination protocol is always consistent. (2) Liveness: If a dynamically generated task \((l, p) \in T\) is assigned to the robot \(r_i\) then it eventually computes consistent \(\Psi_i\).

The multi-robot motion planner described in this section satisfies the following soundness theorem:

**Theorem 2 (Soundness).** If a dynamically generated task \((l, p)\) is assigned to a robot \(r_i\), then the corresponding trajectory \(\xi_i\) computed by MRMP always satisfies the safe-trajectory property \(\Phi_{st}\).

**Proof.** As stated earlier, if \(\xi_i\) is the trajectory computed by the plan-generator using \(\Psi_i\), then it provides the guarantee that consistent\((\Psi_i) \implies (\xi_i \models \Phi_{st})\) and we proved using model-checking that the coordination protocol always satisfies \(\forall \Psi_i, \text{consistent}(\Psi_i)\).

However, MRMP is not complete due to the following reason: for a given task the corresponding robot may not be able to reach the destination due to the fact that its feasible trajectories may be blocked by the other stationary robots.

### 4. VERIFICATION OF DMR SYSTEM

In this section, we describe our approach for verifying that a DMR system \((\mathcal{M})\) satisfies specification \(\Phi\). The system specification \(\Phi\) can be considered as a conjunction of two properties: (1) application specific property \((\Phi_{sp})\) and (2) safe-mission property \((\Phi_{sm})\). For example, in case of the mail delivery system, the property that mails must be delivered in priority order is a part of application specific property whose correctness depends on the implementation of the task-planner, whereas the safe-mission property like drones do not collide, crash are guaranteed by the motion planner. A system satisfies safe-mission property if the trajectories of all the robots always satisfy \(\Phi_{st}\). MRMP implemented in Drona is provably correct (Sec. 3.4) and hence, the system that uses MRMP for planning will satisfy \(\Phi_{sm}\). Therefore, when verifying \(M \models \Phi\), the programmer only needs to prove that \(M \models \Phi_{sp}\).

As explained in Sec. 3.3, the plan executor \((PE)\) processes must step almost-synchronously with symmetric period \(\tau\). All the other processes in the software stack are event-driven and are composed asynchronously. We call the DMR system as a mixed synchronous system as it is a composition of asynchronously composed processes and almost-synchronously composed processes.

#### 4.1 Formal Model of DMR system

We model the DMR mixed synchronous system as a tuple \((k, \mathcal{S}, \mathcal{I}, \mathcal{P}_{sp}, \mathcal{P}_{as}, \chi, \tau, \delta)\) where:

- \(k\) is the number of robots in the system.
- \(\mathcal{S}\) is the set of discrete states of the system which is a product of the local states of all the processes including their communication buffers.
- \(\mathcal{I} \subseteq \mathcal{S}\) is the set of initial states of the system.
- \(\mathcal{P}_{sp} = \{\mathcal{P}_{1, sp}, \mathcal{P}_{2, sp}, \ldots \mathcal{P}_{k, sp}\}\) is the set of process identifiers for the symmetric periodic (PE) processes. \(\mathcal{P}_{sp}\) represents symmetric periodic process running on \(r_i\).
- \(\mathcal{P}_{as} = \{\mathcal{P}_{1, as}, \mathcal{P}_{2, as}, \ldots \mathcal{P}_{k, as}\}\) is the set of process identifiers for the asynchonous processes. \(\mathcal{P}_{as}\) represents composition of asynchronous process running on \(r_i\). \(\mathcal{P}_{as} = TP^m \parallel MRMP^m \parallel SI^m\).
- \(\chi = (\chi_1, \chi_2, \ldots, \chi_k)\) is a vector of real valued local clocks, each robot \(r_i\) has an associated local clock \(\chi_i\).
- \(\delta\) is the common global process timetable for the periodic \(\mathcal{P}_{sp}\) processes. The timetable \(\delta\) is an infinite vector \((\tau^1, \tau^2, \tau^3, \ldots)\) specifying the time instants according to local clock \(\chi_i\) when the process \(\mathcal{P}_{sp}\) executes (steps). In other words, \(\mathcal{P}_{sp}\) makes its \(m\)th step when \(\chi_i(t) = \tau^m\) where \(\chi_i(t)\) is the value of the local clock \(\chi_i\) at global reference time \(t\). Also, since the \(\mathcal{P}_{sp}\) processes step with a period of \(\tau\), \(|\tau^{j+1} - \tau^j| = \tau\)
- \(\delta \subseteq \mathcal{S} \times \Sigma_{MS} \times \mathcal{S}\) is the labeled transition relation for the mixed synchronous system. \(\Sigma_{MS}\) denote \((2^{\mathcal{P}_{sp}} \setminus \{\}) \cup \mathcal{P}_{as}\), the transition labels of the system.

Note that the periodic \(\mathcal{P}_{sp}\) processes have the same timetable but that does not mean that the process step perfectly synchronously, since their local clocks may report different values at the same time \(t\).

**Timed traces:*** A timed trace \(\tau\) of the mixed synchronous system \(\mathcal{M}_{MS}\) is an infinite sequence of the timestamped record of the execution of the system according to the global (ideal) time reference \(t\) and is of the form \(\tau : (s_{0}, t_{0}) \ldots (s_{n}, t_{n}) \ldots\) with \(\forall i, i \geq 0, s_i \in \mathcal{S}, t_i \in \mathbb{R}_{\geq 0}\) and \(t_{i+1} \leq t_i + \delta_{i+1}\) satisfying requirements:

Initiation: \(s_0 \in \mathcal{I}\), and \(\forall i, \chi_i(t_0) = t_0, t_0 = 0\).

Consecution: for all \(i \geq 0\), there is a transition of the form \((s_i, a_i, s_{i+1})\) in \(\delta\) such that the label \(a_i\) is either one of the following:

1. The label \(a_i\) is an asynchronous process, \(a_i \in \mathcal{P}_{as}\) and the transition represents \(a_i\) stepping at time \(t_i\).
2. The label \(a_i\) is a subset of symmetric periodic processes, \(a_i \subseteq \mathcal{P}_{sp}\) and \(\forall j, \mathcal{P}_{sp} \in a_i, \chi_j(t_i) = \tau_m\) for some \(m \in \{0, 1, 2, \ldots\}\). \(\chi_j(t_i)\) is the value of the local clock \(\chi_j\) at current global reference time \(t_i\). This transition represents a subset of symmetric periodic processes making a step whose local clock value at time \(t_i\) is equal to some timetable value. Moreover, \(\mathcal{P}_{sp}\) processes step according to their timetables; thus, if any process \(\mathcal{P}_{sp} \in \mathcal{P}_{sp}\) makes its \(m\)th and \(n\)th steps at times \(t_j\) and \(t_k\) respectively, for \(m < n\), then \(\chi_j(t_j) = \tau_{m_j} < \tau_{n_i} = \chi_i(t_k)\).
4.2 Mixed Synchronous Abstraction

$\mathcal{M}_{MS}$ system described above can be modeled as a hybrid or timed system (due to the continuous dynamics of physical clocks), but the associated methods \cite{11} for verification tend to be less efficient for systems with huge discrete state space. Instead, we construct the discrete abstraction $\hat{\mathcal{M}}_{MS}$ of $\mathcal{M}_{MS}$ that preserves the relevant timing semantics of the ‘mixed synchronous’ systems. We restate the approximate synchrony abstraction introduced in \cite{9} (Theorem 1) for symmetric periodic processes.

**Definition 3.** A system $\mathcal{M}_{as}$ is said to satisfy approximate synchrony (is approximately-synchronous) with parameter $\Delta$ if, for any two processes $P_i$ and $P_j$ in $\mathcal{M}_{as}$, the number of steps $N_i$ and $N_j$ taken by the two processes always satisfies the following condition:

$$|N_i - N_j| \leq \Delta$$

We extend approximate synchrony abstraction to create an untimed mixed synchronous abstraction of $\mathcal{M}_{MS}$.

We define $\hat{\mathcal{M}}_{MS}$ as a tuple $(k, S, I, P_{sp}, P_{as}, \rho_{\Delta}, \delta^a)$ where $\rho_{\Delta}$ is a scheduler process that performs an asynchronous composition of all the processes while enforcing approximate synchrony condition with parameter $\Delta$ only for the $P_{sp}$ processes. The scheduler $\rho_{\Delta}$ maintains counter $N_i$ of the number of steps taken by each process $P_{sp}$ from the initial state. A configuration of $\hat{\mathcal{M}}_{MS}$ is a pair $(s, N)$ where $s \in S$ and $N \in \mathbb{Z}^k$ is the vector of step counts for the $P_{sp}$ processes. The transition function $\delta^a$ for the abstract model $\hat{\mathcal{M}}_{MS}$ can be defined as $((s, N), a_i, (s', N')) \in \delta^a$ iff $\delta(s, a_i, s')$ and one of following holds: (1) $N'_i = N_i + 1$ and $\rho_{\Delta}$ permits all $P_{sp}^j \in a_i$ to make a step, (2) $a_i \in P_{as}$ and $a_i$ makes a step.

$\rho_{\Delta}$ scheduler enforces the mixed synchronous condition during exploration by allowing $P_{sp}$ processes to step iff their step does not violate the approximate synchrony condition (Eq. 7) and the $P_{as}$ are allowed to step.

**Untimed traces:** Traces of $\hat{\mathcal{M}}_{MS}$ are (untimed) sequences of discrete (global) states $s_0, s_1, s_2, \ldots$ where $s_j \in S, s_0 \in I$, and for all $j$, $(s_j, a_j, s_{j+1}) \in \delta^a$.

**Theorem 3.** The abstract model $\hat{\mathcal{M}}_{MS}$ is a sound abstraction of the concrete model $\mathcal{M}_{MS}$. Hence, $\hat{\mathcal{M}}_{MS} \models \Phi$ implies $\mathcal{M}_{MS} \models \Phi$. (proof sketch in appendix)

Note that mixed-synchronous abstraction is critical for the verification of DMR systems. Performing synchronous composition of all processes in the system is unsound and performing asynchronous composition can lead to false-positives due to over-approximation.

5. DRONA FRAMEWORK

In this section, we provide an overview of the DRONA tool-chain. DRONA consists of four main building blocks—a programming language for implementing and specifying a DMR application, a reliable DMR software stack, a model checking backend for efficiently verifying the DMR system, and a runtime library for executing the generated C code on ROS or other robot firmware.

We extend a state-machine based programming language P \cite{7, 3} for the safe programming of DMR applications. A P program comprises of concurrently executing state machines (a.k.a. actors) communicating asynchronously with each other using events. We extend P to ensure that the generated C code from the compiler can be executed on ROS and added features like writing workspace configuration which are specific to DMR programs (more details about P is in Appendix A).

![Figure 3: Drona Tool Chain](image)

A DRONA application consists of four blocks—implementation, specification, workspace config., and test-driver. The implementation is a collection of P state-machines implementing the distributed task-planner. Specifications capture the application specific correctness properties. These specifications are implemented in the form of monitors. Each monitor is a state machine that maintains necessary state for asserting a temporal safety or liveness property. The workspace config XML file provides details about the workspace, like size of the workspace grid, location of static obstacles, location of battery charging points, starting location of each robot, etc. It is used to set up the system and then by the motion planner for synthesizing the motion plan. The test-driver block implements the environment state machines that close the DMR system, which includes machines that nondeterministically pump in task requests in to the system and machines that model discrete behavior of the sensors and robot.

The compiler generates a translation of the DMR application into the ZING modeling language. We extended ZING to support mixed-synchronous abstraction and execution-based model-checking approach to automatically check if the program satisfies the desired properties expressed in the specification block (details about the model-checker implementation is in Appendix D.2).

The compiler also generates C code that is compiled by a standard C compiler and linked against the DRONA runtime and robot SDK to generate the executable code to be deployed on each robot or the ROS simulator.

The generic software stack (Fig. 2) consisting of the MRMP, PE and SI modules is provided as a part of the DRONA tool-chain. We implemented the generic software stack in P and verified its implementation. The entire software stack was implemented in less than 2500
lines of P code. We found and fixed many critical bugs during model-checking phase. The source code for the software stack is available publicly [2] and the detailed links are available in Appendix A.

6. EVALUATION

We empirically evaluate Drona with the following goals: (1) show that the safe plan-generator can be used for on-the-fly motion planning with large number of robots and large workspace size, (2) demonstrate advantages of using Drona for building reliable DMR system by implementing and verifying the priority mail delivery system as a case-study, (3) deploy the generated code from Drona on ROS simulator for various configurations to validate the reliability, and (4) show how time-synchronization error (\(\Delta\)) effects safe optimal path computation.

All experiments were performed on a laptop with 2.5 GHz Intel i7 core processor with 16GB RAM.

**Evaluation of safe plan generator:** Recently, there is an increased interest towards using SMT solvers for motion planning synthesis [15, 22, 24, 25]. The performance of plan generator depends on the complexity of constraints generated, which varies based on the size of workspace, number of robots, their current trajectories, and the density of static obstacles. From our experiments we found that SMT solvers do not scale for plan generation in the context of multi-robot systems. Generating a motion plan with a workspace of size 64x64 and 16 robots takes 2 min 18 sec (see Table 1 implementation details in Appendix C).

We implemented the plan-generator using publicly available A* implementation [1] and encoded the path constraints into A* search. In our evaluation of A* based plan generator, we increase the number of robots from 4 to 128 and consider 2-D grids of sizes 16x16 to 256x256 (our motion planner supports 3-D workspaces, simulation video at [2]).

We generated random workspaces of varying size with obstacle density of 20% (20% of the grid locations are obstacles). We simulated a system with \(n\) robots and created an environment that pumps in a sequence of task requests with random goal location. We measured the amount of time it takes for each robot to compute its trajectory. Table 1 reports the average computation time over 300 invocations of plan-generator for different configurations.

The results show that our plan generator that takes into account time-synchronized clocks is scalable for large grid sizes and number of robots. Hence we believe that it can be used for generating plans on-the-fly in a decentralized fashion with formal guarantees.

**Building mail delivery system:** We implemented the priority mail delivery system in P and composed it with the reliable Drona software stack. We used the mixed synchronous discrete abstraction (Section 4.2) for verifying that the mail delivery system satisfies application specific properties \(\Phi_a\) like (1) mails are always delivered in priority order, (2) mail request if received is eventually delivered, and (3) battery status of the drones is always higher than a safe threshold. These specifications were implemented as P monitors. As Drona supports finite state model-checking, we bounded the environment to nondeterministically send 10 mail delivery requests with random pickup-dropoff locations and verified that the system satisfied \(\Phi_a\). Note that since we are considering finite input, the eventually liveness properties can be considered as safety properties over finite traces. During the process of implementing the generic software stack and the priority mail delivery system we found many critical bugs that would have been hard to find otherwise using traditional simulation based approach. For example, there was a bug (race condition) in the coordination protocol which led to the case where a robot computes its trajectory using an older trajectory of other robot, causing collision. This race condition could not be reproduced with 2 hours of random simulations but was caught in a few seconds using the model-checker.

We deployed the generated code on to a AscTec Firefly drone for conducting simple drone missions of like surveillance of small area around the campus and also performed hardware-in-loop simulations.

**Simulations:** We also implemented a ROS simulator that supports 3-D simulation of the code generated from Drona framework. Simulation videos for various configurations are available at [2]. To validate the reliability of code generated by Drona, we added runtime assertions into the generated C code and ran the simulations for 128 robots with random task generator for 12 hours. We did not find any bug during this stress testing, confirming that the verified code generated from the Drona framework is reliable.

**Effect of \(\Delta\) on planning:** The approximate-synchrony parameter \(\Delta\) represents the clock skew (and thus, step skew) in the system and effects the window of locations avoided by robots when computing trajectory. In other words, it affects how conservative a robot is when computing the trajectory. Hence, the optimal path for a robot may change based on the value of \(\Delta\). A simulation video to demonstrate this scenario is available at [2].

![](http://www.asctec.de/uav-uas-drohnen-flugsysteme/asctec-firefly/)

<table>
<thead>
<tr>
<th>(R)</th>
<th>Grid Size</th>
<th>Computation time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>16x16</td>
<td>0.0174 0.0179 0.0215 0.0518 0.1385</td>
</tr>
<tr>
<td>8</td>
<td>32x32</td>
<td>0.0179 0.0184 0.0249 0.0837 0.2651</td>
</tr>
<tr>
<td>16</td>
<td>64x64</td>
<td>0.0187 0.0206 0.0318 0.0884 0.3038</td>
</tr>
<tr>
<td>32</td>
<td>-</td>
<td>0.0247 0.0435 0.1007 0.3186</td>
</tr>
<tr>
<td>64</td>
<td>-</td>
<td>0.0666 0.1538 0.3882</td>
</tr>
<tr>
<td>128</td>
<td>-</td>
<td>- 0.2293 0.5159</td>
</tr>
</tbody>
</table>

Table 2: Performance of A* based plan-generator

<table>
<thead>
<tr>
<th>(R)</th>
<th>Grid Size</th>
<th>Time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>16x16</td>
<td>0.66 3.5 15.4</td>
</tr>
<tr>
<td>8</td>
<td>32x32</td>
<td>0.9 8.9 33.55</td>
</tr>
<tr>
<td>16</td>
<td>64x64</td>
<td>44.6 138</td>
</tr>
</tbody>
</table>

Table 1: Performance of SMT-based plan-generator


7. RELATED WORK

Related work can be summarized as follows:

**Multi robot motion planning:** The problem of synthesizing collision free trajectories for multi-robot systems in a scenario where the robots are preassigned a set of tasks has been addressed in several prior work. It can be categorized as follows: (1) Centralized motion planning (e.g. [10, 26, 21, 25]) where a central server, given a set of tasks and robots in the system, computes the collision-free trajectory for each robot offline, (2) Decentralized prioritized planning (e.g. [12, 25, 27]) where given a fixed set of tasks, the robots in the system coordinate with each other asynchronously for computing the trajectories. These papers empirically show that decentralized approaches can converge faster than centralized approach. In this paper, we present a decentralized motion planning that can handle dynamically generated tasks and are robust against “almost synchrony”.

**Reactive motion planning:** Recently, there is increased interest towards using temporal logic formalism for synthesizing reactive motion plans [17, 30, 6]. This approach, in principle, can be extended and applied to solve a DMR problem. However, the problem with automated synthesis is that the algorithms scale poorly both with the number of robots and the size of the workspace. Also, they resolve collisions only locally and therefore cannot always guarantee that the resulting motion plan will be deadlock-free and that the robot will eventually reach its destination. Also, in this paper we present a framework that verifies the entire software stack and not just the task-planner and motion-planner.

**Programming models:** Programming frameworks like PTIDES [21] and Giotto [14] have been used for programming and analyzing critical distributed embedded systems software. PTIDES extends the discrete-event model of computation with a carefully chosen relationship between real time and model time for modeling time-synchronized systems. Giotto provides an abstract model for the implementation of periodic soft-ware tasks with hard real-time constraints. The closest related to DRONA is the recently proposed, StarL [20] framework that unifies programming, verification of distributed robotic system. DRONA integrates a state-machine based programming language for event-driven robotics software, it provides a novel motion planner which is robust against clock-ske in distributed systems and present an abstraction based model-checking approach for verification.

8. REFERENCES

[3] P Github. [https://github.com/p-org/P](https://github.com/p-org/P)
APPENDIX

A. DRONA FRAMEWORK DETAILS

Details about the P programming language:
- P Source code: https://github.com/p-org/P

Details about the Drona framework:
- Drona website: https://drona-org.github.io/Drona-Website/
- Drona source code: https://github.com/Drona-Org/Drona
- ROS simulation videos: https://drona-org.github.io/Drona-Website/
- PX4 Demo (building PX4 software using drona): https://www.youtube.com/watch?v=ZzLoNbMChg4

B. THEOREMS

Lemma 1. The abstract model  \( \hat{M}_{MS} \) is a sound abstraction of the concrete model \( M_{MS} \). Hence, \( \hat{M}_{MS} \models \Phi \) implies \( M_{MS} \models \Phi \).

Proof. (Proof Sketch) Let \( \text{traces}(\mathcal{M}) \) represent the set of all “untimed” traces of the system \( \mathcal{M} \). The untiming logic for timed traces is defined by Alur in [a]. \( \mathcal{M} \) is a sound abstraction of \( \mathcal{M} \) if \( \text{traces}(\mathcal{M}) \subseteq \text{traces}(\hat{\mathcal{M}}) \). We derive the proof-sketch using the proof for Theorem 1 in [9] which proves that for a time-synchronized system \( \mathcal{M}_{ps} \) with synchronization \( \beta \), the approximate synchrony based abstract model \( \hat{\mathcal{M}}_{ps} \) is a sound abstraction with parameter \( \Delta = \left\lfloor \frac{2}{7} \right\rfloor \). Informally, \( \text{traces}(\mathcal{M}_{ps}) \subseteq \text{traces}(\hat{\mathcal{M}}_{ps}) \).

Since, \( \mathcal{M}_{MS} \) is a time-synchronized system we know that for the \( P_{sp} \) processes \( \hat{M}_{MS} \) is a sound abstraction of the \( M_{MS} \). Since the \( P_{as} \) are interleaved asynchronously in both \( M_{MS} \) and \( \hat{M}_{MS} \) we can prove that \( \text{traces}(\hat{M}_{MS}) \subseteq \text{traces}(\hat{\mathcal{M}}_{MS}) \).


C. SMT BASED PLAN GENERATOR

In this section, we describe how SMT solver [5] can be used to generate a motion plan for a given motion plan synthesis problem instance. To synthesize a motion plan, we first start with the length of the trajectory \( k \) of robot \( r_i \) to be the \textit{manhattan distance} between the current location of the robot and its goal location ignoring obstacles, and solve the conjunction of the set of constraints Equation (2) – (4) to Equation (6). The constraints are from the theory of linear integer arithmetic and the theory of equality with uninterpreted functions. We represent the obstacles using an uninterpreted function. Current SMT solvers, for example, Z3 [5] can solve the boolean combination of constraints from these theories very efficiently. If there exists a solution of the set of constraints, the solution provides us a motion plan for robot \( r_i \). If no solution exists, we attempt to solve the constraints for a larger value of \( k \). We iterate that process until the value of \( k \) is less than or equal to \( L^{i}_{max} \) (a parameter that represents the maximal length to be considered for generating the trajectory for robot \( r_i \)) or we get a solution of the constraints. As we start our iteration with the value of \( k \) to be the manhattan distance between the start and the goal location (which is the lower bound of the distance between them) and increase the value of \( k \) by 1 in each iteration, a solution of the constraints in an iteration provides a motion plan that generates a trajectory of minimal length. If no motion plan of length less than or equal to \( L^{i}_{max} \) is found, it is guaranteed that there does not exist a feasible motion plan of length less than equal to \( L^{i}_{max} \) for the given problem instance. Once a trajectory of minimal length is found, the next goal is to find the optimal cost trajectory of that length. The optimal cost trajectory is found by introducing a variable total\_cost that represents the total cost of all the primitives used to generate the trajectory and using the \textit{bisection method} in the range of possible values for the total cost of the trajectory.

Implementation of SMT based multi-robot motion planner is available here: https://github.com/Drona-Org/ComplanV2

D. IMPLEMENTATION OF MODEL CHECKING

D.1 Mixed-Synchronous Scheduler

We use ZING [4], an explicit state model checker for the verification of DMR system. ZING supports search prioritization or directed search based on an external scheduler [8]. We implemented the mixed synchrony scheduler (\( \rho_{\Delta} \)) as an external scheduler for ZING that constraints the interleaving explored by model checker during verification. More details about the implementation of model-checking algorithm that uses approximate synchrony scheduler is available in [9] and how to write various external schedulers with ZING is men-
D.2 Execution-based model-checking

When verifying that the DMR system satisfies property $\Phi_a$, the task planner must be composed with a sound model of the motion planner, which given a task returns a safe motion plan satisfying $\Phi_s$ for accomplishing that task. As described in Section 3, computing a safe motion plan involves usage of complex constraint solvers. Hence, creating a sound model of such a motion planner when model checking the system is not possible. To resolve this problem, we use execution based model checking, it is a verification technique based on executing the actual program implementation whenever the sound model is not available during the systematic exploration. We extended Zing, such that during model checking whenever the motion planner invokes $\text{synthesizeMotionPlan}$ function it executes the C implementation of the function which in turn calls the SMT Solver. This technique is sound because same implementation code for $\text{synthesizeMotionPlan}$ is used during model checking and actual execution.

E. EVALUATION OF VERIFICATION

We performed analysis of the application in two phases:

Stratified random sampling: To catch bugs faster in our implementation, we first performed stratified random sampling [8] of executions of the system. In this mode Zing, uniformly samples execution of max depth 100. We were able to find most of the bugs in our implementation during this mode of testing. Note that this is similar to random simulation of systems, but much more scalable as we use a parallel model checker for exploration.

Deterministic exploration: Sampling approaches fail to provide coverage guarantees, for that, we performed deterministic enumeration (using state caching) of all possible executions in the system with max depth 100 and time budget of 10 hours. Table 3 shows the coverage results for various grid sizes and number of robots.

✓ represents that we were able to explore all possible executions till depth 100 and (n) represents explored all possible executions till the depth n in the given time budget.

E.1 Simulations

We performed ROS-based simulations of the system built using DRONA. Snapshot of the simulator is shown below. Simulation videos and the script for running the simulator is available online as mentioned in Sec. A.