DRONA: A Framework for Safe Distributed Mobile Robotics

Ankush Desai
University of California, Berkeley
ankush@eecs.berkeley.edu

Indranil Saha
Indian Institute of Technology, Kanpur
isaha@cse.iitk.ac.in

Jianqiao Yang
University of California, Berkeley
jq.yang@berkeley.edu

Shaz Qadeer
Microsoft Research, Redmond
qadeer@microsoft.com

Sanjit A. Seshia
University of California, Berkeley
sseshia@eecs.berkeley.edu

ABSTRACT

Distributed mobile robotics (DMR) involves teams of networked 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 code generated by the compiler can be executed on platforms such as the robot operating system (ROS).

To demonstrate the efficacy of Drona, we build and verify a 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).

CCS CONCEPTS

•Computing methodologies →Motion path planning; Cooperation and coordination; •Computer systems organization →Robotic autonomy; Embedded software; •Software and its engineering →Software verification and validation;

KEYWORDS

Distributed Robotics, Verification, Programming Language for Robotics, Safe Mobile Robotics, Multi-Robot Motion Planning

ICCPS 2017, Pittsburgh, PA USA

978-1-4503-4965-9/17/04…$$15.00
DOI: http://dx.doi.org/10.1145/3055004.3055022

ACM Reference format:
Ankush Desai, Indranil Saha, Jianqiao Yang, Shaz Qadeer, and Sanjit A. Seshia. 2016. DRONA: A Framework for Safe Distributed Mobile Robotics. In *Proceedings of The 8th ACM/IEEE International Conference on Cyber-Physical Systems, Pittsburgh, PA USA, April 2017 (ICCPS 2017),* 10 pages.
DOI: http://dx.doi.org/10.1145/3055004.3055022

1 INTRODUCTION

Recent demonstrations of autonomous robots collaborating to accomplish complex missions have fueled excitement about 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 in complex safety-critical applications, 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.

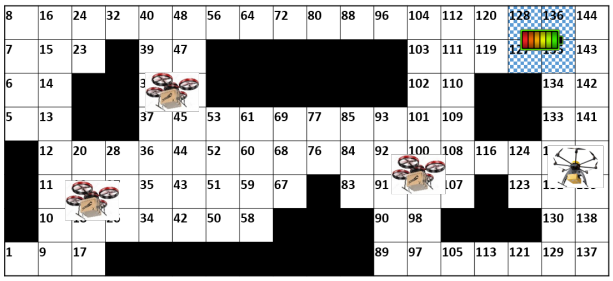

Figure 1: Workspace for the mail delivery system.

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. 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 motion plans 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. [5, 26, 27, 29, 30]) makes an assumption that the robots in the system step synchronously, or in other words, their local clocks are synchronized. However, in distributed systems there is no perfect synchrony and hence this unsound assumption can lead to motion planner computing colliding trajectories. With the advances in time-synchronization protocols [12], clocks in the distributed system can be synchronized within a small bound. One of the salient features of MRMP implemented in DRONA is that it does not assume perfect synchrony of the distributed clocks. It produces safe collision-free trajectories taking into account the "almost synchronized" nature of a time-synchronized DMR system.

A major challenge in programming autonomous reactive robots is to correctly handle nondeterministically generated events and their interleavings. We integrate a state-machine based programming language P [8] 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) [25]. P supports ZING [2, 9], a state-of-the-art model-checker for verification of P programs.

We take a principled approach towards specifying and implementing a *generic* DMR software stack (Sec. 2.1) in P language. A DMR system implemented using DRONA software stack consists of both event-driven asynchronous processes and periodic processes. We formalize such a DMR system as a *mixed-synchronous* system. We verify the system using ZING's implementation of a model checking approach based on the notion of *approximate synchrony*, an idea we previously introduced [10].

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 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 [11].

In summary, our contributions are the following:

- 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.
- 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.
- 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. Using DRONA, we found several critical bugs in our implementation which a rigorous random simulation based approach failed to find.

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

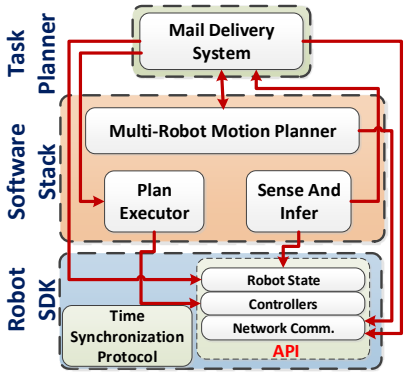

Figure 2: Robotics Software Stack

Fig. 2 presents the modular software stack executed by each robot in the DMR system. 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.

The plan-executor (PE) module ensures 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-planer only if it infers an event that requires task planner's attention. For example, there is a *Battery-Monitor* state-machine that monitors the battery sensor data-stream

coming from the robot and only informs the task-planner when the battery level is less than a threshold. Robot manufacturing companies also provide a software development kit (SDK) that implements basic primitives for programmatically controlling a robot, sampling its state and communicating with other robots in the system. We verified the implementation of DMR software stack under the assumption that the robot SDK is correct. Further details about the software stack are available online [11].

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 Ω. The set of free locations in the workspace is denoted by F, where $F = W \setminus \Omega$. The fixed set of robots operating in the workspace is denoted by the set $R = \{r_1, \ldots, r_{|R|}\}$.

Tasks: In a DMR application, tasks can be generated dynamically and assigned to a robot. An atomic task is denoted as the tuple (l, p), where $l \in F$ denotes the goal location where the robot needs to reach for finishing the task, and $p \in \mathbb{N}$ denotes the unique identifier of the task. We denote by T the set of all atomic tasks. A complex task can be represented as a sequence of atomic tasks. In the rest of the paper, we will use the term *task* to refer to an atomic task.

Motion primitives: Motion primitives are a set of short closed-loop trajectories of a robot under the action of a set of precomputed control laws [20, 22]. The set of motion primitives form the basis of the motion for a robot. A robot moves from its current location to a destination location by executing a sequence of motion primitives. We denote by Γ the set of all motion primitives available for a robot. For example, in the most simple case a ground robot has five motion primitives: $\{\mathsf{H}, \mathsf{L}, \mathsf{R}, \mathsf{U}, \mathsf{D}\}$, where the primitive H keeps the robot in the same grid block and the primitives L, R, U and D move the robot to the adjacent left, right, upper, and lower grid block respectively.

For a grid location l and a motion primitive $\gamma \in \Gamma$, we denote by $\mathtt{post}(l, \gamma)$ the location where the robot moves when the motion primitive γ is applied at l. We use $\mathtt{intermediate}(l, \gamma)$ to denote the set of locations through which the robot may traverse after applying γ at location l (including l and $\mathtt{post}(l, \gamma)$). For a motion primitive $\gamma \in \Gamma$, we denote by $\mathtt{cost}(\gamma)$ the cost (e.g., energy expenditure) to execute the motion primitive. We assume that for all robots in the system, each motion primitive requires τ 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 2.1 (Motion Plan). A motion plan is defined as a sequence of motion primitives to be applied to a robot r_i to move from its current location l_c^i to a goal location l_g^i. 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 trajectory of a robot r_i can be represented as a sequence of timestamped locations $(\tau_0^i, l_0^i), (\tau_1^i, l_1^i) \ldots$, where τ_n^i represents the n-th periodic time step for robot r_i. In the rest of the paper we refer to (τ_n^i, l_n^i) as l_n^i representing the location of robot r_i in the n-th time step. The size of the period $|\tau_n^i - \tau_{n+1}^i| = \tau$, where τ is the time it takes to execute any motion primitive.

Definition 2.2 (Trajectory). Given the current location l_c^i 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 τ_n^i, the trajectory of the robot is a sequence of locations $\xi_i = (l_n^i l_{n+1}^i \ldots l_{n+k}^i)$, such that $l_n^i = l_c^i$, $\forall q \in \{0, \ldots, k-1\}$, γ_{q+1} is applied to the robot at location l_{n+q}^i at the time step τ_{n+q}^i and $l_{n+q+1}^i = \mathtt{post}(l_{n+q}^i, \gamma_{q+1})$.

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

Motion planning problem in DMR:

Problem 1. *Given a set of robots $R = \{r_1, \ldots, r_{|R|}\}$ operating in a common workspace W, if a dynamically generated task $(l, p) \in \mathsf{T}$ is assigned to a robot $r_i \in R$, find trajectory ξ_i such that it satisfies safe task-completion property Φ_{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 (Ψ), synthesize a safe trajectory that is robust against time-synchronization errors in distributed systems and satisfies Φ_{st} (Sec. 3.2).

3.1 Distributed Trajectory Coordination

In a DMR system, tasks are generated dynamically. Hence, the motion planner for such a system should be able to compute 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 the P programming language.

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 an 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 (*more details about P language is available at [24]*).

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 ($currTask_{id}$ and l_g^i) 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_c^i to robot 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} stores identifiers of the robots that have sent their trajectories as a response to the `ReqForCurrentTraj` event, and Ψ_i stores 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 ζ_j to the set Ψ_i. The planner state machine is blocked in CoordinateAndGeneratePlan state until it receives `CurrentTraj` event from all the robots.

On receiving trajectories from all the robots (line 19), the planner invokes the `synthesizeMotionPlan` function with its current location l_c^i, the goal location l_g^i, the set of static obstacles Ω and the set of trajectories of all the robots Ψ_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 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 each tasks. If the planner of r_i receives a `ReqForCurrentTraj` event from r_j in the CoordinateAndGeneratePlan state and if the task identifier $task_{id}$ in the event is less than its current task identifier $currTask_{id}$ then it implies that the robot r_j is dealing with a higher priority task. In such a case, the motion planner of r_i sends its current location l_c^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_{pend}, and once it computes its own trajectory, sends the trajectory to unblock r_j (Line 23-25).

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

Notice that if the planner for robot r_i generates trajectory ξ_i, then ξ_i is always safe as the coordination protocol guarantees that

Protocol 1 Decentralized Motion Planner

```
1:  machine DecentralizedMotionPlanner {
2:      start state WaitForTaskRequest {
3:          entry { l_c^i ←getCurrentLocation() }
4:          on NewTask (task : T) do {
5:              currTask_id ← task.id, l_g^i ← task.goal
6:              goto CoordinateAndGeneratePlan
7:          }
8:          on ReqForCurrentTraj (task_id, r_j) do {
9:              send (r_j, CurrentTraj, (r_i, [l_c^i]))
10:         }
11:     }
12:     state CoordinateAndGeneratePlan {
13:         entry {
14:             R_pend ← {} , R_recv ← {}, Ψ_i ← {}
15:             broadcast (ReqForCurrentTraj, (currTask_id, r_i))
16:         }
17:         on CurrentTraj (r_j, ζ_j) do {
18:             R_recv ← R_recv ∪ {r_j}, Ψ_i ← Ψ_i ∪ {ζ_j}
19:             if (sizeof(R_recv) = |R|) then
20:                 ρ_i ← synthesizeMotionPlan(l_c^i, l_g^i, Ω, Ψ_i)
21:                 SendMotionPlanToPlanExecutor(ρ_i)
22:                 ξ_i ← ConvertMotionPlanToTraj(ρ_i)
23:                 foreach r_j ∈ R_pend
24:                     send (r_j, CurrentTraj, (r_i, ξ_i))
25:                 end
26:                 goto WaitForPlanCompletion
27:             end if
28:         }
29:         on ReqForCurrentTraj (task_id, r_j) do {
30:             if (task_id ≤ currTask_id) then
31:                 send (r_j, CurrentTraj, (r_i, [l_c^i]))
32:             else
33:                 R_pend ← R_pend ∪ {r_j}
34:             end if
35:         }
36:     }
37:     state WaitForPlanCompletion {
38:         on ReqForCurrentTraj (task_id, r_j) do {
39:             send (r_j, CurrentTraj, (r_i, ξ_i))
40:         }
41:         on Reset () do {
42:             goto WaitForTaskRequest
43:         }
44:     }
45: }
```

all future trajectories computed by any other robot r_j will have ξ_i in Ψ_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 task-completion property Φ_{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_c^i), the goal location (l_g^i), the set of static obstacles (Ω), and the set of current trajectories of other robots (Ψ_i). We call the tuple $\mathcal{P}_i = \langle l_c^i, l_g^i, \Omega, \Psi_i \rangle$ as the *motion plan synthesis problem* instance for robot r_i.

Recall that a trajectory ξ_i of robot r_i is a sequence of locations $(l_n^i, l_{n+1}^i, \ldots, l_{n+k}^i)$, where the trajectory starts at the n-th time step. We adopt a technique based on composition of motion primitives [26, 27] to solve the motion-plan synthesis problem. To generate such a trajectory ξ_i, we must synthesize a motion plan (Def. 2.1) $\rho_i = (\gamma_1, \gamma_2, \ldots, \gamma_k)$, where $\gamma_q \in \Gamma$, $1 \leq q \leq k$. Recollect that the desired trajectory (Def. 2.2) is realized by applying motion primitive γ_{q+1} to the robot at time step τ_{n+q}^i.

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 Γ, and the time step τ_n^i when the plan executor will start executing the motion plan, synthesize a motion plan $\rho_i = (\gamma_1 \ldots \gamma_k)$ such that the trajectory $\xi_i = (l_n^i l_{n+1}^i \ldots l_{n+k}^i)$ generated by the plan executor by executing the motion plan ρ_i satisfies the safe task-completion property Φ_{st}.*

Accounting for clock skew: Each robot $r_i \in R$ operates based on its own local clock χ_i. Let t denote an ideal global time reference (just for purposes of formalization). We denote by $\chi_i(t)$ the valuation of the clock χ_i 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 ϕ_c. We assume that the DMR software stack implements a time-synchronization protocol [12] 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 *perfect synchrony*. Otherwise, the clocks are almost-synchronous with precision $\beta > 0$.

To capture the skew between timed trajectories of two robots, we define a parameter Δ that denotes the maximum offset between the sequences of periodic steps τ^i and τ^j of any two robots r_i and r_j. The value of Δ is computed as $\Delta = \left\lceil \frac{\beta}{\tau} \right\rceil$.

Theorem 3.1. *If the local clocks of robots r_i and r_j are time-synchronized with a synchronization precision β, and at some global time point t, if robot r_i takes the time step τ_p^i and robot r_j takes the time step τ_q^j, then $|p - q| \leq \Delta$, where Δ is given by $\Delta = \left\lceil \frac{\beta}{\tau} \right\rceil$ where τ is the duration of a time step [10].*

The above condition is called *approximate synchrony* and was introduced and proved in our previous work [10]. Informally, Theorem 3.1 states that if the clocks of two robots are synchronized within a bound β then the difference between the number of periodic steps taken by the two robots is bounded by Δ. 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 Δ 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 $\rho_i = (\gamma_1, \ldots, \gamma_k)$ is synthesized from a motion plan synthesis problem instance $\mathcal{P}_i = \langle l_c^i, l_g^i, \Omega, \Psi \rangle$. 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 `post`, `cost`, and `intermediate` used in this section are defined in Sec. 2.2.

The objective function is given as follows:

$$\underset{(\gamma_1, \gamma_2, \ldots, \gamma_k)}{\texttt{minimize}} \sum_{j=1}^{k} \texttt{cost}(\gamma_j) \tag{1}$$

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

(1) Initial and final location: The first location in ξ_i is the current location l_c^i of the robot. Similarly, the last location in ξ_i must be the goal location l_g^i.

$$l_n^i = l_c^i \wedge l_{n+k}^i = l_g^i \tag{2}$$

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

$$\forall q \in \{0, \ldots, k-1\} : \; l_{n+q+1}^i \in \texttt{post}(l_{n+q}^i, \gamma_{q+1}) \tag{3}$$

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

$$\forall q \in \{0, \ldots, k-1\} \; \forall l \in \texttt{intermediate}(l_{n+q}^i, \gamma_{q+1}) : l \notin \Omega \tag{4}$$

This constraint ensures the obstacle avoidance component ϕ_o of the *safe task-completion* property Φ_{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 $\xi_i = (l_n^i l_{n+1}^i \ldots l_{n'}^i)$ with the trajectories of other robots captured in the set Ψ. The trajectory of any other robot r_j is denoted by $(l_m^j, \ldots, l_n^j, \ldots, l_{m'}^j) \in \Psi$, where $m \leq n$.

The following constraint guarantees collision avoidance property ϕ_c for a perfectly synchronous system:

$$
\begin{aligned}
&\forall r_j \in R \setminus \{r_i\}, (l^j_m, \ldots, l^j_n, \ldots, l^j_{m'}) \in \Psi : \\
&((\forall q \in \{n, \ldots, \texttt{min}(n', m')\} : \ l^i_q \neq l^j_q) \wedge \\
&/^* \text{ The robot } r_i \text{ reaches destination before robot } r_j \ ^*/ \\
&(n' < m' \Rightarrow \forall q \in \{n'+1, \ldots, m'\} : l^i_{n'} \neq l^j_q) \wedge \\
&/^* \text{ The robot } r_i \text{ reaches destination after robot } r_j \ ^*/ \\
&(n' > m' \Rightarrow \forall q \in \{m'+1, \ldots, n'\} : l^i_q \neq l^j_{m'}))
\end{aligned} \tag{5}
$$

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 β. We do so using the notion of approximate synchrony introduced in Theorem 3.1. Specifically, to ensure collision avoidance with another robot, the plan synthesizer of a robot should ensure that its location at time step τ^i_n does not overlap with the location of the other robot *at any step in the range of* $(\tau^i_n - \Delta, \tau^i_n + \Delta)$. Eq. (6) extends Eq. (5) to encode collision avoidance constraint with an approximate synchrony bound of Δ.

$$
\begin{aligned}
&\forall r_j \in R \setminus \{r_i\}, (l^j_m, l^j_{m+1}, \ldots, l^j_n, \ldots, l^j_{m'}) \in \Psi : \\
&\quad ((\forall q \in \{n, \ldots, \texttt{min}(n', m')\} \ \forall p \in \{q - \Delta, \ldots, q + \Delta\} : \\
&\qquad (n \leq p \leq m' \Rightarrow l^i_q \neq l^j_p) \wedge \\
&\qquad (p < m \Rightarrow l^i_q \neq l^j_m) \ \wedge \ (p > m' \Rightarrow l^i_q \neq l^j_{m'})) \wedge \\
&/^* \text{ The robot } r_i \text{ reaches destination before robot } r_j \ ^*/ \\
&((n' < m') \Rightarrow \forall q \in \{n'+1, \ldots, m'\} \ \forall p \in \{q - \Delta, \ldots, q + \Delta\} : \\
&\qquad (p \leq n' \Rightarrow l^i_p \neq l^j_q) \ \wedge \ (p > n' \Rightarrow l^i_{n'} \neq l^j_q)) \wedge \\
&/^* \text{ The robot } r_i \text{ reaches destination after robot } r_j \ ^*/ \\
&((n' > m') \Rightarrow \forall q \in \{m'+1, \ldots, n'\} \ \forall p \in \{q - \Delta, \ldots, q + \Delta\} : \\
&\qquad (p \leq m' \Rightarrow l^i_q \neq l^j_p) \ \wedge \ (p > m' \Rightarrow l^i_q \neq l^j_{m'})))
\end{aligned} \tag{6}
$$

SMT solver based safe plan-generator: To synthesize the motion plan using an satisfiability modulo theories (SMT) solver [4], we first start by initializing the length of the trajectory (k) to be the *manhattan distance* between the current location of the robot and its goal location. The constraints (Eq.(1)-Eq.(6)) are from the theory of linear integer arithmetic and the theory of equality with uninterpreted functions. We represent the obstacles using an uninterpreted function. If there exists a solution for the set of constraints, the solution provides us the desired motion plan. If no solution exists, we increase the value of k by 1 and attempt to solve the constraints again. 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). 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.

However, as our experimental results reveal (Sec. 6), an SMT 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 [16] 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 encode 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 ρ_i sends it to the plan-executor module. The plan-executor executes the sequence of motion-primitives in ρ_i such that the robot r_i realizes its timed-trajectory ξ_i (Def. 2.2). It is implemented as a periodic state-machine with the duration of each period as τ, 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 τ, i.e, $\forall r_i \in R, \forall n, |\tau^i_n - \tau^i_{n+1}| = \tau$. Since path-executor at each robot r_i step using its local clock χ_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

Recollect that when computing a trajectory for a robot r_i, the execution of MRMP is decomposed into two phases: first the coordination protocol computes the avoid trajectories set Ψ_i which is then used by the safe plan-generator for computing the collision-free trajectory ξ_i. We say that the avoid trajectories set Ψ_i is *consistent* if $\forall \zeta_j \in \Psi_i, \zeta_j = \xi_j$, where ζ_j is the trajectory sent by robot r_j to robot r_i and ξ_j is the actual trajectory being executed by robot r_j.

As described in Sec. 3.2.2, the A* based plan-generator always generates trajectories that satisfy the safe task-completion property Φ_{st} under the assumption that avoid trajectory set Ψ_i is *consistent*. In other words, given the set of trajectories Ψ_i, if the plan-generator computes trajectory ξ_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 Ψ_i computed by the coordination protocol is always *consistent*. (2) *Liveness:* If a dynamically generated task $(l, p) \in \mathsf{T}$ is assigned to the robot r_i then it eventually computes *consistent* Ψ_i.

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

Theorem 3.2 (Soundness). *If a dynamically generated task (l_g^i, p) is assigned to a robot r_i then the corresponding trajectory ξ_i computed by* MRMP *always satisfies the safe task-completion property Φ_{st}.*

Proof. As stated earlier, if ξ_i is the trajectory computed by the plan-generator using Ψ_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 $\forall \Psi_i, consistent(\Psi_i)$. □

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 Φ.

As explained in Sec. 3.3, for the robots in the system to successfully follow their computed trajectories, the plan executor (PE) processes must step *almost-synchronously* with symmetric period τ. Hence, the PE processes across robots are implemented as periodic processes. All the other processes in the software stack e.g., TP, MRMP, and SI 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 $(k, \mathcal{S}, \mathcal{I}, \mathcal{P}_{sp}, \mathcal{P}_{as}, \vec{\chi}, \tau, \delta)$ 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.
- $\mathcal{I} \subseteq \mathcal{S}$ is the set of initial states of the system.
- $\mathcal{P}_{sp} = \{\mathcal{P}_{sp}^1, \mathcal{P}_{sp}^2, \ldots, \mathcal{P}_{sp}^k\}$ is the set of process identifiers for the symmetric periodic (PE) processes. $\mathcal{P}_{sp}^i$ represents symmetric periodic process running on r_i.
- $\mathcal{P}_{as} = \{\mathcal{P}_{as}^1, \mathcal{P}_{as}^2, \ldots, \mathcal{P}_{as}^k\}$ is the set of process identifiers for the asynchronous processes. $\mathcal{P}_{as}^i$ represents composition of asynchronous process running on r_i. $\mathcal{P}_{as}^i = TP^i \parallel MRMP^i \parallel SI^i$.
- $\vec{\chi} = (\chi_1, \chi_2, \ldots, \chi_k)$ is a vector of real valued local clocks, each robot r_i has an associated local clock χ_i.
- $\vec{\tau}$ is the common global process timetable for the periodic $\mathcal{P}_{sp}$ processes. The timetable $\vec{\tau}$ is an infinite vector $(\tau^1, \tau^2, \tau^3, \ldots)$ specifying the time instants according to local clock χ_i when the process $\mathcal{P}_{sp}^i$ executes (steps). In other words, $\mathcal{P}_{sp}^i$ makes its jth step when $\chi_i(t) = \tau^j$ where $\chi_i(t)$ is the value of the local clock χ_i at global reference time t. Also, since the $\mathcal{P}_{sp}$ processes step with a period of τ, $|\tau^{j+1} - \tau^j| = \tau$.
- $\delta \subseteq \mathcal{S} \times \Sigma_{MS} \times \mathcal{S}$ is the labeled transition relation for the *mixed synchronous* system. Σ_{MS} denote $(2^{\mathcal{P}_{sp}} \setminus \{\}) \sqcup \mathcal{P}_{as}$, 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 processes step perfectly synchronously, since their local clocks may report different values at the same global time t.

Timed traces: A timed trace σ 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 $\sigma : (s_0, t_0), \ldots (s_n, t_n) \ldots$ with $\forall i.\ i \geq 0$, $s_i \in \mathcal{S}$, $t_i \in \mathbb{R}_{\geq 0}$ and $t_i \leq t_{i+1}$ satisfying requirements:

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

Consecution: for all $i \geq 0$, there is a transition of the form (s_i, a_i, s_{i+1}) in δ 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 process 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 $\forall j. \mathcal{P}_{sp}^j \in a_i$, $\chi_j(t_i) = \tau^m$ for some $m \in \{0, 1, 2, \ldots\}$. $\chi_j(t_i)$ is the value of the local clock χ_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 $\mathcal{P}_{sp}^i \in \mathcal{P}_{sp}$ makes its mth and lth steps at times t_j and t_k respectively, for $m < l$, then $\chi_i(t_j) = \tau^{m_i} < \tau^{l_i} = \chi_i(t_k)$.

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 [14, 19] for verification tend to be less efficient for systems with huge discrete state space. Instead, we construct the discrete abstraction $\widehat{\mathcal{M}}_{MS}$ of $\mathcal{M}_{MS}$ that preserves the relevant timing semantics of the '*mixed synchronous*' systems.

We restate the *approximate synchrony* abstraction introduced in [10] (Theorem 3.1) for symmetric periodic processes.

Definition 4.1. A system $\mathcal{M}_{as}$ is said to satisfy *approximate synchrony* (is *approximately-synchronous*) with parameter Δ if, for any two processes $\mathcal{P}_i$ and $\mathcal{P}_j$ in $\mathcal{M}_{as}$, the number of steps N_i and N_j taken by the two processes always satisfies the following condition:

$$|N_i - N_j| \leq \Delta \tag{7}$$

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

We define $\widehat{\mathcal{M}}_{MS}$ as a tuple $(k, \mathcal{S}, \mathcal{I}, \mathcal{P}_{sp}, \mathcal{P}_{as}, \rho_\Delta, \delta^a)$ where ρ_Δ is a scheduler process that performs an asynchronous composition of all the processes while enforcing approximate synchrony condition with parameter Δ (computed using Theorem 3.1) only for the $\mathcal{P}_{sp}$ processes. The scheduler ρ_Δ maintains counter N_i of the number of steps taken by each process $\mathcal{P}_{sp}^i$ from the initial state. A configuration of $\widehat{\mathcal{M}}_{MS}$ is a pair (s, N) where $s \in \mathcal{S}$ and $N \in \mathbb{Z}^k$ is the vector of step counts for the $\mathcal{P}_{sp}$ processes. The transition function δ^a for the abstract model $\widehat{\mathcal{M}}_{MS}$ 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) $N_j' = N_j + 1$ and ρ_Δ permits all $\mathcal{P}_{sp}^j \in a_i$ to make a step, (2) $a_i \in \mathcal{P}_{as}$ and a_i makes a step.

ρ_Δ scheduler enforces the mixed synchrony condition during exploration by allowing $\mathcal{P}_{sp}$ processes to step iff their step does not violate the approximate synchrony condition and the $\mathcal{P}_{as}$ are always allowed to step.

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

Theorem 4.2. *The abstract model $\widehat{\mathcal{M}}_{MS}$ is a sound abstraction of the concrete model $\mathcal{M}_{MS}$. Hence, $\widehat{\mathcal{M}}_{MS} \models \Phi$ implies $\mathcal{M}_{MS} \models \Phi$.*

Proof. (Proof Sketch) Let $traces(\mathcal{M})$ represent the set of all *untimed* traces of the system $\mathcal{M}$. The untiming logic for timed traces is as defined by Alur in [1]. $\hat{\mathcal{M}}$ is a sound abstraction of $\mathcal{M}$ if $traces(\mathcal{M}) \subseteq traces(\hat{\mathcal{M}})$ We derive the proof-sketch from Theorem 3.1 in [10] which proves that for a time-synchronized system $\mathcal{M}_{ps}$ with synchronization β, the approximate synchrony based abstract model $\hat{\mathcal{M}}_{ps}$ is a sound abstraction with parameter $\Delta = \left\lceil \frac{\beta}{\tau} \right\rceil$. Since the $\mathcal{P}_{as}$ are interleaved asynchronously in both $\mathcal{M}_{MS}$ and $\widehat{\mathcal{M}}_{MS}$ we can further prove that $traces(\mathcal{M}_{MS}) \subseteq traces(\widehat{\mathcal{M}}_{MS})$. □

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.

Implementation of the verification approach: Zing model checker supports directed search based on an external scheduler [9]. We implemented the mixed synchrony scheduler (ρ_Δ) as an external scheduler for Zing that constraints the interleaving explored during verification. The model-checking algorithm that uses approximate synchrony scheduler is described in [10].

5 DRONA FRAMEWORK

The Drona tool-chain consists of four main building blocks — (1) an *event-driven programming language* for implementing and specifying a DMR application, (2) a reliable DMR *software stack*, (3) a *model checking* backend for efficiently verifying the DMR system, and (4) a *runtime library* for executing the generated C code on ROS.

We extended the state-machine based programming language P [8, 24] so that the generated C code from the compiler can be directly executed on ROS. We also extended the language with primitives for specifying the workspace configuration.

A Drona application implemented using the extended P language consists of four blocks—*implementation*, *specification*, *workspace config.*, and *test-driver*. The *implementation* block is a collection of P state-machines implementing the task planner (TP) module. *Specification* block capture the application specific correctness properties. These specifications are implemented in the form of *monitors* and can be used for any 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. The *test-driver* block implements the *finite* environment state machines that close the DMR system for verification.

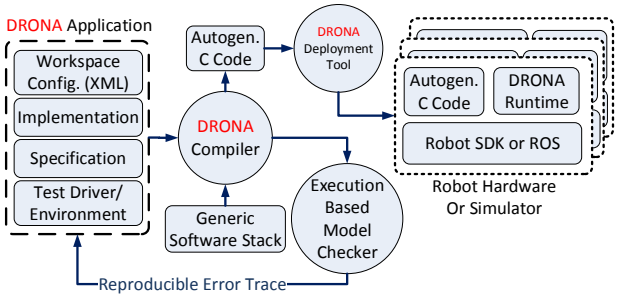

Figure 3: Drona Tool Chain

The Drona compiler generates a translation of the DMR application into the Zing modeling language. We extended Zing to support *mixed-synchronous* abstraction to automatically check if the program satisfies the desired properties expressed in the specification block. 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. The entire software stack was implemented in less than 2500 lines of P code and was systematically tested using the Zing model-checker.

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 the ROS simulator for various configurations to validate the reliability, and
(4) Show how time-synchronization error (Δ) 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 plan synthesis [23, 26, 27]. 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 the state-of-the-art solver Z3 [6] does 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 secs (see Table 1).

We implemented the plan-generator using a publicly available A* implementation [3] 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

$\|R\|$	Time in seconds		
	Grid Size		
	16x16	32x32	64x64
4	0.66	3.5	15.4
8	0.9	8.5	33.55
16	-	44.6	138

Table 1: Performance of SMT-based plan-generator

256x256 (our motion planner supports 3-D workspaces, simulation video at [11]). We generated random workspaces of varying size such that 20% of the grid locations are occupied by 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 2 reports the average computation time over 300 invocations of plan-generator for different configurations.

$\|R\|$	Computation time in seconds				
	Grid Size				
	16x16	32x32	64x64	128x128	256x256
4	0.0174	0.0179	0.0215	0.0518	0.1485
8	0.0179	0.0184	0.0249	0.0837	0.2651
16	0.0187	0.0206	0.0318	0.0884	0.3038
32	-	0.0247	0.0435	0.1007	0.3186
64	-	-	0.0666	0.1538	0.3882
128	-	-	-	0.2293	0.5159

Table 2: Performance of A* based plan-generator

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 Φ_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 Φ_a. 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 also deployed the generated code on to an AscTec Firefly[1] drone for conducting simple drone missions.

[1]http://www.asctec.de/uav-uas-drohnen-flugsysteme/asctec-firefly/

Evaluation of Verification: We performed analysis of the application in two phases:

(1) Stratified random sampling: To catch shallow bugs in our implementation, we first performed stratified sampling [9] of executions. In this mode, Zing uniformly samples execution of max depth 1000. We were able to find most of the bugs in our implementation during this mode of testing. Note that this is similar to performing random simulations, but much more scalable as we use a parallel model checker for exploration.

$\|R\|$	Max depth explored in 10 hours		
	Grid Size		
	8x8	16x16	32x32
2	✓	✓	✓
4	✓	✓	✓
8	✓	✓	(78)

Table 3: Scalability of verification approach

(2) Deterministic exploration: Sampling based approaches fail to provide coverage guarantees, for that, we performed deterministic enumeration (with 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 Zing explored all possible executions till depth 100 and (n) represents that Zing explored all possible executions till the depth n in the given time budget.

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 [11]. 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 Δ on planning: The approximate-synchrony parameter Δ 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 Δ. A simulation video to demonstrate this scenario is available at [11].

7 RELATED WORK

Related work can be summarized into the following categories:

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. [13, 26–28]) 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. [5, 15, 29]) 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 [7, 18, 31]. 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 Giotto [17] have been used for building critical distributed embedded systems software. Giotto provides an abstract model for the implementation of periodic software tasks with real-time constraints. The closest work related to Drona is the recently proposed StarL [21] framework, that unifies programming, specification and 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-skew in distributed systems and presents an abstraction based model-checking approach for verification.

8 CONCLUSION

In this paper, we presented the Drona software framework for building reliable DMR systems. The multi-robot motion planner (MRMP) implemented in Drona is provably correct and scales efficiently for large number of robots and large workspaces. MRMP is the first to take into account the time-synchronization error in a distributed multi-robot system when generating safe motion plans . Using a model-checking approach leveraging the notion of approximate synchrony, we were able to find bugs in our implementation which rigorous random simulations failed to find.

As future work, we plan on applying the Drona software stack for real-world complex missions. We also plan to extend Drona with runtime monitoring and adaptation.

ACKNOWLEDGMENTS

The first and last authors were supported in part by the TerraSwarm Research Center, one of six centers supported by the STARnet phase of the Focus Center Research Program (FCRP) a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

REFERENCES

[1] Rajeev Alur and David L Dill. 1994. A theory of timed automata. *Theoretical computer science* 126, 2 (1994), 183–235.

[2] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Xie. 2004. Zing: A Model Checker for Concurrent Software. In *16th International Conference on Computer Aided Verification (CAV)*.

[3] Astar. 2017. Astar Algorithm Cpp Github. https://github.com/justinhj/astar-algorithm-cpp.git. (2017).

[4] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. 2009. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, Armin Biere, Hans van Maaren, and Toby Walsh (Eds.). Vol. 4. IOS Press, Chapter 8.

[5] Michal Cáp, Peter Novák, Martin Selecký, Jan Faigl, and Jiff Vokffnek. 2013. Asynchronous decentralized prioritized planning for coordination in multi-robot system. In *International Conference on Intelligent Robots and Systems*. IEEE, 3822–3829.

[6] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 337–340.

[7] Jonathan A. DeCastro, Javier Alonso-Mora, Vasu Raman, Daniela Rus, and Hadas Kress-Gazit. 2015. Collision-Free Reactive Mission and Motion Planning for Multi-Robot Systems. In *International Symposium on Robotics Research (ISRR)*. Sestri Levante, Italy.

[8] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. 2013. P: Safe Asynchronous Event-driven Programming. In *Programming Language Design and Implementation (PLDI)*. 321–332.

[9] Ankush Desai, Shaz Qadeer, and Sanjit A. Seshia. 2015. Systematic Testing of Asynchronous Reactive Systems. In *Foundations of Software Engineering (FSE)*. 73–83.

[10] Ankush Desai, Sanjit A. Seshia, Shaz Qadeer, David Broman, and John C. Eidson. 2015. Approximate Synchrony: An Abstraction for Distributed Almost-Synchronous Systems. In *Computer Aided Verification (CAV)*. 429–448.

[11] Drona. 2017. Drona Website. https://drona-org.github.io/Drona/. (2017).

[12] John Eidson and Kang Lee. 2002. IEEE 1588 standard for a precision clock synchronization protocol for networked measurement and control systems. In *Sensors for Industry Conference, 2002. 2nd ISA/IEEE*. Ieee, 98–105.

[13] Michael Erdmann and Tomas Lozano-Perez. 1986. On Multiple Moving Objects. *Algorithmica* 2 (1986), 1419–1424.

[14] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. 2011. SpaceEx: Scalable verification of hybrid systems. In *Computer Aided Verification (CAV)*. 379–395.

[15] Yi Guo and L. E. Parker. 2002. A distributed and optimal motion planning approach for multiple mobile robots. In *International Conference on Robotics and Automation (ICRA)*, Vol. 3. 2612–2619.

[16] P. E. Hart, N. J. Nilsson, and B. Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transaction on Systems Science and Cybernetics* (1968).

[17] Thomas A Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. 2001. Giotto: A time-triggered language for embedded programming. In *International Workshop on Embedded Software*. Springer, 166–184.

[18] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. 2009. Temporal-logic-based reactive mission and motion planning. *IEEE transactions on robotics* 6 (2009), 1370–1381.

[19] Kim G Larsen, Paul Pettersson, and Wang Yi. 1997. UPPAAL in a nutshell. *International journal on software tools for technology transfer* 1, 1-2 (1997), 134–152.

[20] Steven M LaValle. 2006. *Planning algorithms*. Cambridge university press.

[21] Yixiao Lin and Sayan Mitra. 2015. StarL: Towards a Unified Framework for Programming, Simulating and Verifying Distributed Robotic Systems. In *Languages, Compilers and Tools for Embedded Systems (LCTES)*. Article 9, 10 pages.

[22] Daniel Mellinger and Vijay Kumar. 2011. Minimum snap trajectory generation and control for quadrotors. In *International Conference on Robotics and Automation (ICRA)*. 2520–2525.

[23] Srinivas Nedunuri, Sailesh Prabhu, Mark Moll, Swarat Chaudhuri, and Lydia E Kavraki. 2014. SMT-based synthesis of integrated task and motion plans from plan outlines. In *International Conference on Robotics and Automation (ICRA)*. IEEE, 655–662.

[24] P. 2017. P Github. https://github.com/p-org/P. (2017).

[25] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. 2009. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*.

[26] Indranil Saha, Rattanachai Ramaithitima, Vijay Kumar, George J Pappas, and Sanjit A Seshia. 2014. Automated composition of motion primitives for multi-robot systems from safe LTL specifications. In *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 1525–1532.

[27] Indranil Saha, Rattanachai Ramaithitima, Vijay Kumar, George J Pappas, and Sanjit A Seshia. 2016. Implan: scalable incremental motion planning for multi-robot systems. In *International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 1–10.

[28] Jur P Van Den Berg and Mark H Overmars. 2005. Prioritized motion planning for multiple robots. In *Intelligent Robots and Systems (IROS)*. IEEE, 430–435.

[29] Prasanna Velagapudi, Katia Sycara, and Paul Scerri. 2010. Decentralized prioritized planning in large multirobot teams. In *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 4603–4609.

[30] Glenn Wagner and Howie Choset. 2011. M*: A complete multirobot path planning algorithm with performance bounds. In *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 3260–3267.

[31] Tichakorn Wongpiromsarn, Ufuk Topcu, and Richard M Murray. 2012. Receding horizon temporal logic planning. *IEEE Trans. Automat. Control* 57, 11 (2012), 2817–2830.