Programming Distributed Cyber-Physical Systems

Ritwika Ghosh

University Of Illinois at Urbana-Champaign

1 Introduction

Distributed autonomous robots are being used for mapping [**?**] and delivery services [**?**], and have enormous potential in transforming industries such as manufacturing [**?**,**?**], transportation [**?**,**?**], agriculture [**?**,**?**]. Following the trends in cloud, mobile, and machine learning applications, programmability is key in unlocking this potential, as robotics platforms become more open and hardware developers shift to the applications marketplace. Programming languages like C#, Swift, Python, and development tools like LLVM [**?**] have

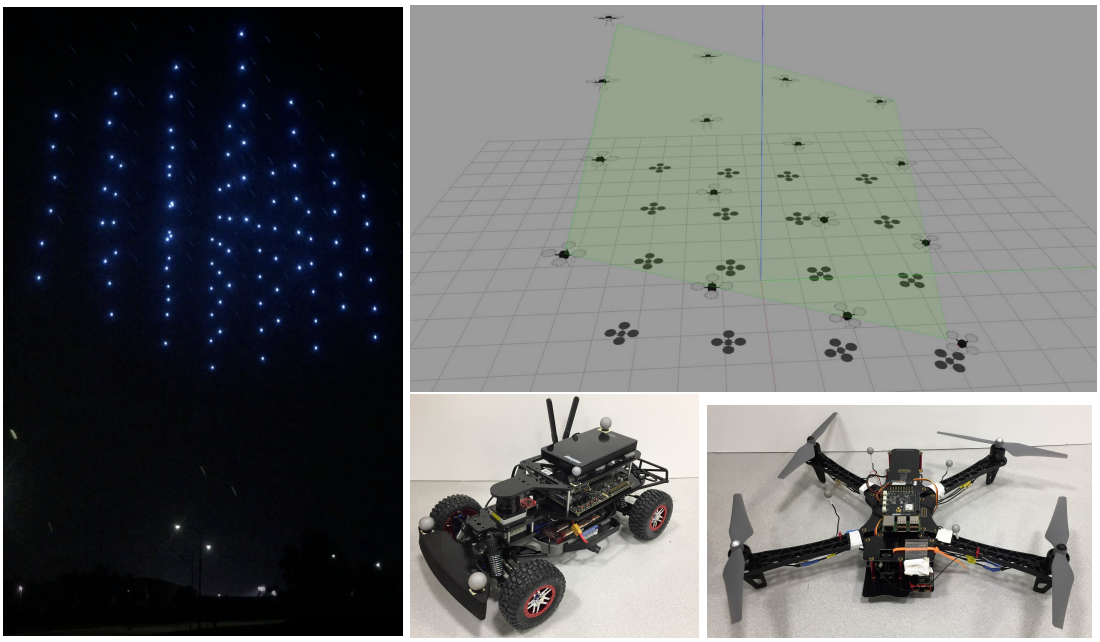

Fig. 1. Swarm formation show by FireFly Inc. (*Left*). Simulation of shape formation (*Top Right*). Cars and quadcopters we can deploy and test *Koord* programs.

helped make millions of people, with diverse backgrounds, into mobile application developers, and open source software libraries like PyTorch [**?**] and Tensorflow [**?**] have propelled the surge in machine learning research and development. To a lesser degree, similar efforts in democratization of roboticsa are being made. Among existing robotics programming frameworks, ROS [**?**] provideS hardware abstractions, device drivers, messaging protocols, many common library functions and has become widely used. Libraries such as PyRobot [**?**] and PythonRobotics [**?**] provide hardware-independent implementations of common functions for physical manipulation and navigation of individual robots. Nevertheless, it requires significant effort and time of the order of weeks to develop, simulate, and debug a new application for a single mobile robot—not including the effort to build the robot hardware. The required effort grows quickly for distributed and heterogeneous systems, as none of the existing robotics libraries provide either (a) support for distributed coordination, or (b) easy portability of code across different platforms. Further, available domain specific languages (DSL) for robotics are tightly coupled with platforms, and they combine low-level sensing, communication, and control tasks with the application-level logic. This tight-coupling and the attendant lack

of abstraction hinders application development on all fronts—portability, code reuse, and verification and validation (V&V).

In particular, formal reasoning about a collection of robots communicating, coordinating, and interacting with a physical environment is complexified by cyber-physical interactions. Correctness under concurrency and asynchrony are prominent research problems in distributed computing. Correctness under noise, disturbances, and imprecise platform (plant) models is studied intensively by roboticists and control theorists. The analysis techniques from these communities are based on very different formal models and mathematics, and both would be necessary to provide satisfactory safety guarantees for distributed robotic applications. Further, mathematical models and formal analysis techniques at the application design level often do not take into account the gap between design and implementation.

One of the motivations for my thesis is *not* to combine all of the above in an *all encompassing formalism*; but to create a language that separates the concerns of building a reliable distributed cyber-physical application to divide and conquer using existing analyses from both the robotics and formal methods communities. The other, is to empirically demonstrate the feasibility of implementing such a language, and narrow the gap between mathematical modelling and implementation as much as possible.

Contributions I have designed a formal semantics of a high level language [**?**], which provides several abstractions for separation of platform-dependent and independent concerns, and implemented an executable semantics of the same in the $\mathbb{K}$ semantic framework. *Koord* applications can be simulated or deployed using the CyPhyHouse toolchain, which includes a compiler for *Koord*, a high fidelity simulator, deployment and monitoring tools [**?**]. I have also developed a tool, using a verification approach that requires defined port assumptions, proof obligations to provide guarantees which can be posed as inductive invariants [**?**]. While I have only considered safety properties or invariants in *Koord* applications, my work on verifying self-stabilisation of distributed algorithms [**?**] provides an insight into extending the aforementioned verification approach to progress properties.

2 Overview and Approach

Consider the application LineForm in Figure **??**, written in the high level programming language *Koord*. LineForm implements a simple formation control protocol of the type used for drone shows like the one seen in Figure **??**. LineForm makes an arbitrary number of robots (drones) line up uniformly between two extremal robots. The programming tools I have built to compile and deploy *Koord* code on a heterogeneous fleet of robotic platforms; these tools can help automate the verification of these types of applications by allowing decomposition of the proofs into platform dependent and independent proof obligations. Finally, the *Koord* simulator can help find violations of assumptions made for verification.

```
1  using Motion:
2    sensors: Point position
3    actuators: Point target
4
5  allread: Point x[N_SYS]
6  init:
7    x[pid] = Motion.position
9  TargetUpdate:
10   pre: (pid == N_SYS − 1 or pid == 0)
11   eff:
12       Motion.target = (x[pid+1] + x[pid−1]) / 2
13       x[pid] = Motion.position
```

Fig. 2. *Koord* program LineForm for a set of robots to form a line.

Koord is designed as a high-level, event-driven language in which application programs use *shared variables* for coordination across robots and *ports* for interaction with hardware-specific subroutines, In a dis-

tributed robotics setting, instances of the same *Koord* program are executed by each participating robot to solve problems collectively.

Modules and port abstractions. The application programs in my design of *Koord* interact with the sensors and low-level controllers of the robot platform through reads from *sensor* and writes to *actuator* ports(*variables*). For example, LineForm uses a *module* (library) called *Motion* which provides a sensor port called *position* that publishes the robot's position, and an actuator port called *target* for specifying a target position. Thus, these ports provide an abstraction over various possible sensor and controller implementations and environments. Implementations of the modules are part of the *Koord runtime system* and they implement hardware specific functions. For example, the CyPhyHouse toolchain uses an implementation of the *Motion* module for a quadcopter, relying on an indoor camera based positioning system to update the *position* port, and it uses an RRT based [**?**] path planner and motion controller. The *Motion* module abstraction is implemented for a small racing vehicle platform using the same indoor positioning system but a different pid controller.

2.1 Semantics and invariant properties

I have developed the full semantics of *Koord* using $\mathbb{K}$ [**?**]. The execution semantics of any applications for multi-robot systems are complicated by issues of asynchrony, consistency of shared memory, and interactions between software and the physical environment. The $\mathbb{K}$ rewriting engine makes the formal language semantics *executable*, and enables exhaustive exploration of non-deterministic behaviors of *Koord* applications.

For LineForm, a natural requirement is to restrict all robots to stay within a certain safe area, at all times (Geofencing). More precisely, given a (hyper)rectangle $rect(x_{min}, x_{max})$ defined by its two corners x_{min} and x_{max}, if all robots are initialized within the rectangle, then all robots should always stay in the rectangle. This requirement can be stated as:

Invariant 1 $\bigwedge_{i \in ID} \begin{pmatrix} M.pos_i \in rect(x_{min}, x_{max}) \\ \wedge\ x[i] \in rect(x_{min}, x_{max}) \end{pmatrix}$ where $M.pos$ is the shorthand for $Motion.position$.

An invariant like the above can be established in two steps: first, assuming that all the robot positions are in $rect(x_{min}, x_{max})$, we show that the targets computed by LineForm are also in $rect(x_{min}, x_{max})$. The $\mathbb{K}$ semantics of *Koord* enables construction of the symbolic post states of the *TargetUpdate* event and prove Invariant **??** using the *Koord* prover (Figure **??**) I built.

The second step is to show that for any robot, assuming that the computed targets are in $rect(x_{min}, x_{max})$, the controller implementing *Motion* module indeed keeps the robot inside $rect(x_{min}, x_{max})$. For this step, one has to reason about how each robot moves when its implementation of the *Motion* module is given a *target*. *Koord* helps identify and decompose the overall proof into assumptions that the *Module* implementations need to guarantee.

For example, one can state the key assumption needed for Invariant **??** as:

Port Assumption 1

$$\forall t \in [0, \delta], f(M.pos, M.tgt, t) \subseteq rect(M.pos, M.tgt),$$

where $M.tgt$ is the shorthand for $Motion.target$, f is a function giving the position of the robot at time t, moving to $M.tgt$, from $M.pos$. This assumption states that the robot's *Motion* module should ensure that it is moving within the bounding rectangle between its position and target within the duration of a round. These types of assumptions about the control system can be discharged using verification engines for reasoning about continuous behavior of dynamical systems.

2.2 Simulation based assumption validation

Assumption **??** may appear benign at a glance, but it may be violated in some conditions. Using the high-fidelity *Koord* simulator, a designer can gain insights about when such assumptions are violated. The

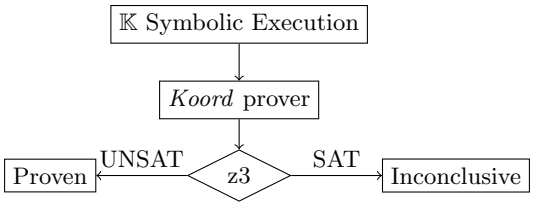

Fig. 3. 𝕂 semantics based invariant checking for *Koord*.

simulator executes complied *Koord* code together with detailed physical models for the robots, ROS-based interactions with sensor models, and UDP-based message passing.

Users can also use the simulator to detect early in the development process if the assumptions for correctness are too strong under specific scenarios, and revise the assumptions iteratively. Using the *Koord* simulator in conjunction with other verification tools, one can discover that Assumption **??** is violated in three rather common scenarios: First, if a robot has to avoid obstacles, then it may have to go around the obstacle and hence out of the bound. Second, the assumption fails for robots with nonholonomic dynamics such as wheeled robots. Third, the inertia of the robot may force it go out of bound temporarily.

2.3 Compilation and deployment.

In addition to the formal language, semantics, analysis, and simulation, the complete CyPhyHouse tool chain [**?**] includes compilation and deployment to heterogeneous platforms including drones and race cars. Once developers install the CyPhyHouse ROS [**?**] based run-time libraries (middleware) on a platform and provide a device specific configuration denoting the mapping from *Koord* module ports to low level sensor and actuator ROS messages, the port based abstraction (module) then allows the same *Koord* program to run on this platform. The modular structure of CyPhyHouse *middleware* I have built will make it easy for a roboticist to add support for new hardware platforms.

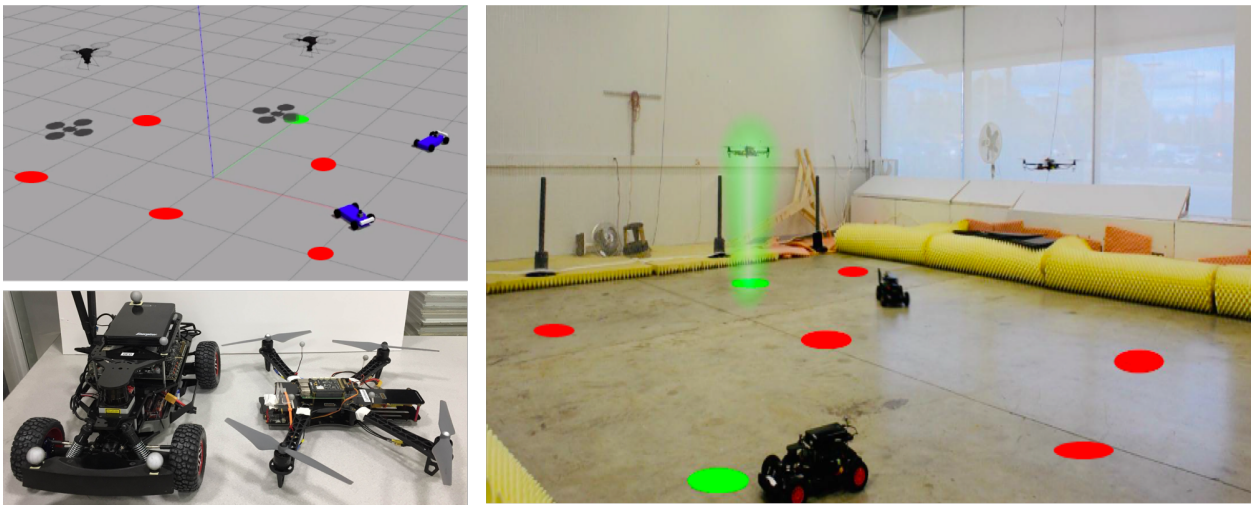

Fig. 4. *Right:* Annotated snapshot of a distributed task allocation application deployed on four cars and drones using the CyPhyHouse toolchain in the IRL test arena. The red tasks are incomplete, and the green are completed. *Left bottom:* different robotic platforms: the F1/10 Car and the quadcopter. *Left top:* Visualization of the same application running in the CyPhyHouse simulator which interfaces with *Gazebo*.

3 *Koord* language design

In this section, we present the syntax and the semantics of *Koord*. When a *Koord* application is deployed on a fleet of `N_SYS` robots, each robot runs an instance of the same program. There is a known set of identifiers $\texttt{ID} = \{0, 1, \ldots, \texttt{N\_SYS} - 1\}$, and each robot is assigned a unique index $\texttt{pid} \in \texttt{ID}$. The execution of the *Koord* program advances in a synchronous, *round-by-round* fashion, where each round lasts for δ time, and $\delta > 0$ is a platform specific execution parameter. During this period, the robots compute, move, and communicate with each other through distributed shared memory.

3.1 Syntax

Figure **??** shows the partial grammar of *Koord* syntax. Each robot program consists of (a) declarations of the interfaces between the program and the sensor/actuator modules, (b) declarations of shared and local program variables, and (c) events, consisting of preconditions and effects. Robot programs (rule *Program*) first can import sensor/actuator modules. The module import grammar production specifies the interfaces or ports: it contains all input and output ports for actuators (*APorts*) and sensors (*SPorts*) that the program uses. To summarize, there are following three types of names for reading/writing values:

(i) *Sensor and actuator ports* are used to read from sensor ports and write to actuator ports of controllers.
(ii) *Local program variables* record the state of the program.
(iii) *Distributed shared variables* are used for coordination across robots. All shared variables can be read by all participating robots; an **allwrite** variables can be written by any participating robot; while an **allread** variables can be written only by a single-writer.

User can then optionally specify a statement to set the initial values of program variables (rule *Init*). The main body of the program is a sequence of events (rule *Event*) which include a Boolean **pre**condition and an **eff**ect. The effect of an event is also a statement (rule *Effect*). We skip the syntax rules for statements, expressions, data types, and functions due to the page limit. A statement (rule *Stmt*) in *Koord* resembles those in most imperative languages and includes conditional statements, function calls, assignments, blocks of statements, atomic statements for mutual exclusion, etc. Mutual exclusion is always an essential feature when shared variables are involved. *Koord* provides a locking mechanism using the keyword **atomic** to update the shared variable safely. The user can also define functions and abstract data types (tuples of the basic data types).

In the syntax presented in Figure **??**, given an nonterminal NT, $NT^{?}$ means that it is optional in the syntax at that position, NT^{*} refers to zero or more occurrences, and NT^{+} refers to one or more occurrences. The expression $(E1 \mid E2)$ denotes that one can use either $E1$ or $E2$.

3.2 Configurations

The semantics of a *Koord* program execution is based on synchronous rounds divided into *event transitions* and *environment transitions* that update the *system configuration*. In each round, each robot performs at most one event. The update performed by a single robot executing an event is modeled as an instantaneous transition that updates the program variables; however, different events executed by different robots may interleave in an arbitrary order. In between the events of successive rounds, $\delta > 0$ duration of time elapses, the program variables remain constant while the values held by the sensor and actuator ports may change. These are modeled as environment transitions that advance time as well as the sensor and actuator ports. Thus, each round consists of a burst of (at most `N_SYS`) event transitions followed by an environment transition. This is a standard model for synchronous distributed systems where the speed of computation is much faster than the speed of communication and physical movement [**?**,**?**].

We now describe the system state, or *system configurations* which we use to formalize *Koord* semantics.

```
Program    ::= Def* Import* DeclBlock Init? Event+
Def        ::= TypeDef — FuncDef
Import     ::= using identifier : SPorts APorts
SPorts     ::= sensors : VarDecl+
APorts     ::= actuators : VarDecl+

DeclBlock  ::= AWDecl? ARDecl? LocalDecl?
AWDecl     ::= allwrite : VarDecl+
ARDecl     ::= allread : ARVarDecl+
LocalDecl  ::= local : VarDecl+
VarDecl    ::= Type identifier — Type identifier = Val
ARVarDecl  ::= Type identifier [ N_SYS ]

Init       ::= init : Stmt
Event      ::= identifier : Precond Effect
Precond    ::= pre : Expr
Effect     ::= eff : Stmt

Stmt       ::= atomic Stmt — Stmt StmtList — ...
```

Fig. 5. Partial *Koord* syntax rules.

System configurations. A *system configuration* is a tuple $\boldsymbol{c} = (\{L_i\}_{i\in\mathtt{ID}}, S, \tau, \mathit{turn})$, where

(i) $\{L_i\}_{i\in\mathtt{ID}}$ or $\{L_i\}$ in short is an indexed set of *robot configurations*–one for each participating robot. L_i refers to the configuration of the i-th element, i.e., the i-th robot in the system.
(ii) $S : \mathit{Var} \mapsto \mathit{Val}$ is the *global context*, mapping all shared variable names to their values.
(iii) $\tau \in \mathbb{R}_{\geq 0}$ is the *global time*.
(iv) $\mathit{turn} \in \{\mathtt{prog}, \mathtt{env}\}$ is a binary *bookkeeping* variable determining whether program or environment transitions are being processed.

Bookkeeping variables are invisible in the language syntax, and only used in the semantics.

Robot configurations. A *robot configuration* is used to specify the semantics of each robot. As a *Koord* program is run on a system of robots, each participating robot would have its own set of module ports and local variables, along with a local copy of each shared variable. Given a *Koord* program P, we can define *Var* be the set of variables, *Val* be the set of values that an expression in *Koord* can evaluate to, *CPorts* be the set of sensor and actuator ports of the controller being used, and *Events* the set of events in P. A robot configuration is a tuple $L = (M, \mathit{cp}, \mathit{turn})$, where

(i) $M : \mathit{Var} \mapsto \mathit{Val}$ is its *local context* mapping both local and shared variables to values. Note that this implies M includes a copy of shared variable values.
(ii) $\mathit{cp} : \mathit{CPorts} \mapsto \mathit{Val}$ is the mapping of sensor and actuator ports to values.
(iii) $\mathit{turn} \in \{\mathtt{prog}, \mathtt{env}\}$ is a bookkeeping variable indicating whether this robot should be executing a program or environment transition.

For readability, we use the dot (".") notation to access components of a robot configuration L. For example, $L.M$ means accessing the local context M in the tuple L.

Black-box functions for environment transitions To define the executable $\mathbb{K}$ semantics of *Koord* applications, we have to provide executable descriptions for the environment transitions. The type of this executable object (f) is defined by *CPorts*, namely, $f : [\mathit{CPorts} \mapsto \mathit{Val}] \times \mathbb{R}_{\geq 0} \mapsto [\mathit{CPorts} \mapsto \mathit{Val}]$. That is, given old sensor and actuator values and a time point, f should return the new values for all sensor and actual ports. Depending on whether we have an explicit or a black-box model for f, the executable semantics will enable different types of analysis as we shall see in Section **??**.

3.3 Semantics

$$\frac{\langle S, L, St\rangle \rightarrow_{stmt} \langle S', L', St'\rangle}{\langle S, L, St\ StList\rangle \rightarrow_{stmt} \langle S', L', St'\ StList\rangle}\ \textsc{StmtSeq1}$$

$$\langle S, L, \cdot\ StList\rangle \rightarrow_{stmt} \langle S, L, StList\rangle\ \textsc{StmtSeq2}$$

$$\frac{x \in Keys(S) \wedge x \in Keys(L.M) \wedge L'.M = L.M[x \mapsto v]}{\langle S, L, x = v\rangle \rightarrow_S \langle S[x \mapsto v], L', \cdot\rangle}\ \textsc{SvarAssign}$$

$$\frac{x \notin Keys(S) \wedge x \in Keys(L.M) \wedge L'.M = L.M[x \mapsto v]}{\langle S, L, x = v\rangle \rightarrow_S \langle S, L', \cdot\rangle}\ \textsc{LvarAssign}$$

Fig. 6. Example statement level semantic rules for *Koord*.

We will describe only the interesting semantic rules of *Koord* above the event level. Rule LvarAssign and Rule SvarAssign showing the semantic rules for local and shared variable assignment respectively are provided as examples of statement level rules. Rule StmtSeq1 and StmtSeq2 show how a statement representing a sequence of statements is executed. More details on statement and expression level semantics will be available in a future extended version of the paper.

$$\langle S, L, \oplus\rangle \rightarrow_{stmt} \langle S, L, \cdot\rangle\ \textsc{SkipEvent}$$

$$\langle S, (M, cp, \texttt{prog}), \cdot\rangle \rightarrow_{stmt} \langle S, (M, cp, \texttt{env}), \cdot\rangle\ \textsc{EndEvent}$$

$$\frac{\forall x \in Keys(S), M' = M[x \mapsto S[x]] \wedge cp' = f(cp, \delta)}{\langle S, (M, cp, \texttt{env})\rangle \rightarrow_{env} \langle S, (M', cp', \texttt{prog})\rangle}\ \textsc{RobotEnv}$$

Fig. 7. Partial per robot semantic rules for *Koord*.

Per robot semantics. First, we present the semantics of executing an event for each robot, which will help us discuss the semantics of the whole system. All rules for statement semantics are of type

$$\rightarrow_{stmt} \subseteq (\mathbb{S} \times \mathbb{L} \times (Stmt \cup \{\oplus, \cdot\})) \mapsto \wp(\mathbb{S} \times \mathbb{L} \times Stmt \cup \{\cdot\}),$$

where *Stmt* refers to the set of all possible statements allowed by *Koord* syntax. This relation takes as input a tuple of (1) a global context, (2) a robot configuration, and (3) a statement, and maps it to a set of such tuples.

The symbols '$\oplus$' and '$\cdot$' are not in *Koord* but internal syntactic structures. '$\oplus$' is to denote nondeterministic selection of events, and '$\cdot$' is to indicate an "empty" statement.

Rule SelectEvent in Figure **??** shows that any event may be executed when the precondition *Cond* is evaluated to true, and by replacing $\oplus$ with the event effect *Body*, it ensures only one event is selected and executed. The event effect is then executed following the semantics of each statement in *Body*. Rule SkipEvent allows the robot to skip the event completely. At the end of the event, the sequence of statements becomes empty '$\cdot$'. Rule EndEvent then makes sure the *turn* of the robot is set to `env` indicating that an environment transition will occur afterwards.

Similarly, we define the semantics of how each robot interacts with environment including other robots. The environment transition rule is of type

$$\rightarrow_{env} \subseteq (\mathbb{S} \times \mathbb{L}) \mapsto \wp(\mathbb{S} \times \mathbb{L}),$$

which takes a global context and a robot configuration as input. Rule RobotEnv simply states that the new local context M' is the old local context M updated with the global context S; thus ensuring that all robots have consistent shared variable values before the next program transition. New sensor readings cp' is then obtained by evaluating the black-box dynamics f with time δ. In an actual execution, the controller would run the program on hardware, whose sensor ports evolve for δ time between program transitions. This formalization allows the ports to behave arbitrarily over δ-transitions. Hence in verification, additional assumptions over the behavior of the sensor and actuator ports are needed. Finally, the *turn* of the robot is set back to `prog`.

System semantics. With the event semantic for each robot, we can then define the execution for the distributed *Koord* program. The rewrite rule is a mapping from an initial system configuration to a set of configurations. It has the type

$$\rightarrow_G \subseteq \mathbb{C} \mapsto \wp(\mathbb{C}),$$

where $\mathbb{C}$ is the set of all possible system configurations.

Rule EventTrans expresses that starting from a system configuration $\boldsymbol{c} = (\{L_i\}, S, \tau, \texttt{prog})$, a robot i with the configuration L_i starts by selecting an enabled event, executes the event via a sequence of $\rightarrow_{stmt}$ rewrites, and sets its own *turn* to `env` at the end of the event execution. The system goes from a configuration $\boldsymbol{c}$ to $\boldsymbol{c}' = (\{L_i'\}, S', \tau, \texttt{prog})$, with possibly different robot configurations and global context depending on whether any statement executed resulted in writes to shared variables. The system can display nondeterministic behaviors arising from different robots executing their events in different orders. After all robots enter the `env` turn, rule EndProgTrans sets the global *turn* from `prog` to `env` indicating the end of program transition, and an environment transition will occur afterwards.

$$\frac{\begin{array}{c}\exists i \in \texttt{ID}, \langle S, L_i, \oplus\rangle \rightarrow_{stmt} \langle S', L_i', \cdot\rangle \\ \wedge\ L_i.turn = \texttt{prog} \wedge L_i'.turn = \texttt{env}\end{array}}{(\{L_i\}, S, \tau, \texttt{prog}) \rightarrow_G (\{L_i'\}, S', \tau, \texttt{prog})}\ \textsc{EventTrans}$$

$$\frac{\forall i \in \texttt{ID}, L_i.turn = \texttt{env}}{(\{L_i\}, S, \tau, \texttt{prog}) \rightarrow_G (\{L_i\}, S, \tau, \texttt{env})}\ \textsc{EndProgTrans}$$

$$\frac{\begin{array}{c}\forall i \in \texttt{ID}, \langle S, L_i\rangle \rightarrow_{env} \langle S, L_i'\rangle \\ \wedge\ L_i.turn = \texttt{env} \wedge L_i'.turn = \texttt{prog}\end{array}}{(\{L_i\}, S, \tau, \texttt{env}) \rightarrow_G (\{L_i'\}, S, \tau + \delta, \texttt{prog})}\ \textsc{EnvTrans}$$

Fig. 8. System semantic rules for *Koord*.

Rule EnvTrans shows the semantics of the system configuration after rule EndProgTrans. This rule synchronizes the environment transitions of each robot and ensure that the global time τ advances to $\tau + \delta$.

3.4 Synchronization and consistency

Following our semantic rules in Section **??**, careful readers would notice that all event transitions of *Koord* program takes *zero* time. The environment transitions however take δ time for the evolution of the sensor and actuator ports together with the update of the local context from the global context. In this section, we discuss how this semantic abstracts the behavior of a distributed cyber-physical system, and how our tool chain in [**?**] provides a faithful implementation of such abstraction.

To reiterate, the following are the timing requirements from rule EventTrans and EnvTrans: (a) an event transition takes *zero* time, (b) new values of controller ports are sampled at the end of each round (c) shared variables should reach consistent values within δ time, and (d) a global clock is used to synchronize each δ-time round. The first two requirements are achievable if the time taken to complete a program transition is negligible compared to δ, and δ can be a common multiple of the sampling intervals of all controller ports in use. These constraints are reasonable when computation and communication is comparatively much faster. For *Motion* module as an example, our position sensor on each device publishes every 0.01 sec (100Hz) while the CPU on each drone is 1.4 GHz. If we set δ to be 0.01 sec, an event transition taking 10K CPU cycles is still less than 0.1% of δ.

Requirement (**??**) and (**??**) are common research questions in distributed computing with an extensive literature. A global clock can be achieved with existing techniques that synchronize all local clocks on robots. In [**?**], we use message passing to implement distributed shared memory for shared variables. We ensure that the time taken to propagate values through messages and reach consistency is smaller than δ, and the update is visible in the next round of program transitions for all robots. We therefore conclude our round based semantic with shared memory is a reasonable abstraction.

4 Verifying *Koord* programs

We have built the semantics of *Koord* in the $\mathbb{K}$ framework to enable decoupled analyses of platform-independent discrete part and the platform-dependent (dynamic) parts of distributed multi-robot systems. The *events* in an *Koord* program define the discrete computations in the system. The effect of a robot i executing event $e \in \mathit{Events}$ on a configuration $\boldsymbol{c} \in \mathbb{C}$, can be seen as a $\rightarrow_{\mathit{stmt}}$ application to $\langle \boldsymbol{c}.S, \boldsymbol{c}.L_i, \mathit{Body}\rangle$, where e is *"eventName:* **pre:** *Cond* **eff:** *Body"*.

4.1 Reachable configurations

Given a set of system configurations $\mathcal{C}$, we define the following functions using the semantic rules of Section **??**: (i) $\mathit{Post}_e(\boldsymbol{c}, i)$ returns the set of configurations obtained by robot i executing event $e \in \mathit{Events}$ from a configuration $\boldsymbol{c}$. (ii) $\mathit{Post}(\mathcal{C}, i)$ returns the set of configurations obtained by robot i executing any event from a configuration in $\mathcal{C}$. (iii) $\mathit{Post}(\mathcal{C}, \boldsymbol{p})$ returns all configurations visited, when robots execute their events in the order $\boldsymbol{p}$, where $\boldsymbol{p}$ is a sequence of $p_i \in \mathtt{ID}$. (iv) $\mathit{Post}(\mathcal{C})$ is the union of $\mathit{Post}(\mathcal{C}, \boldsymbol{p})$ over all orders $\boldsymbol{p}$. (v) $\mathit{End}_{\mathtt{prog}}(\mathcal{C})$ is the set of configurations reached from $\mathcal{C}$ *after* a program transition.

$$
\begin{aligned}
\mathit{Post}_e(\boldsymbol{c}, i) &:= \{\boldsymbol{c}' \mid \mathit{Cond}_{\boldsymbol{c}.S, \boldsymbol{c}.L_i} \\
&\qquad \wedge \langle \boldsymbol{c}.S, \boldsymbol{c}.L_i, \mathit{Body}\rangle \rightarrow_{\mathit{stmt}} \langle \boldsymbol{c}'.S, \boldsymbol{c}'.L_i, \cdot\rangle\}, \\
\mathit{Skip}(\boldsymbol{c}, i) &:= \{\boldsymbol{c}' \mid \langle \boldsymbol{c}.S, \boldsymbol{c}.L_i, \cdot\rangle \rightarrow_{\mathit{stmt}} \langle \boldsymbol{c}'.S, \boldsymbol{c}'.L_i, \cdot\rangle\} \\
\mathit{Post}(\mathcal{C}, i) &:= \bigcup_{\boldsymbol{c} \in \mathcal{C}} \left(\mathit{Skip}(\boldsymbol{c}, i) \cup \bigcup_{e \in \mathit{Events}} \mathit{Post}_e(\boldsymbol{c}, i) \right), \\
\mathit{Post}(\mathcal{C}, \boldsymbol{p}) &:= \begin{cases} \emptyset, & \text{if } \boldsymbol{p} = () \\ \mathit{Post}(\mathit{Post}(\mathcal{C}, p_0), \boldsymbol{p}'), & \text{if } \boldsymbol{p} = (p_0, \boldsymbol{p}') \end{cases} \\
\mathit{Post}(\mathcal{C}) &:= \bigcup_{\boldsymbol{p} \in \mathit{perms}(\mathtt{ID})} \mathit{Post}(\mathcal{C}, \boldsymbol{p}), \\
\mathit{End}_{\mathtt{prog}}(\mathcal{C}) &:= \{\boldsymbol{c} \in \mathit{Post}(\mathcal{C}) \mid \forall i \in \mathtt{ID}, \boldsymbol{c}.L_i.\mathit{turn} = \mathtt{env}\},
\end{aligned}
$$

In the above, a sequence $\boldsymbol{p} = (p_0, \boldsymbol{p}')$, is written as a concatenation of the first element p_0 and the suffix $\boldsymbol{p}'$. Also, $\mathit{perms}(\mathtt{ID})$ refers to the set of permutations of $\mathtt{ID}$.

Next, we define the configurations that are reached during and after an environment transition. Recall that environment transitions capture the evolution of the actuator ports over a time interval $[0, \delta]$—all other

parts of the configuration remain unchanged. Our *Koord* semantics defines the environment transitions with a *parameter* which is a (possibly black-box) function that captures the dynamics of individual robots.[1] Given such a function f_i for each robot i, we define the function $traj : \mathbb{C} \times [0, \delta] \mapsto \mathbb{C}$ to represent the evolution of the system over a $[0, \delta]$ time interval. *traj* is constructed by simply update all controller ports *cp* of all robots with their f_i. That is,

$$\boldsymbol{c}' = traj(\boldsymbol{c}, t) \Leftrightarrow \begin{pmatrix} \forall i \in \texttt{ID}, \boldsymbol{c}'.L_i.cp = f_i(\boldsymbol{c}.L_i.cp, t) \\ \wedge\ \boldsymbol{c}'.L_i.M = \boldsymbol{c}.L_i.M \\ \wedge\ \boldsymbol{c}'.L_i.turn = \boldsymbol{c}.L_i.turn \\ \wedge\ \boldsymbol{c}'.S = \boldsymbol{c}.S \wedge \boldsymbol{c}'.\tau = \boldsymbol{c}.\tau \\ \wedge\ \boldsymbol{c}'.turn = \boldsymbol{c}.turn \end{pmatrix}$$

Notice those additional constraints making sure all other fields of $\boldsymbol{c}$ and $\boldsymbol{c}'$ stay the same. We will use $\wedge \ldots$ to skip these kind of constraints in the later sections for simplicity. The set of system configurations $Pt_{[t_1,t_2]}(\mathcal{C})$ reached in an interval $[t_1, t_2]$:

$$Pt_{[t_1,t_2]}(\mathcal{C}) := \{\boldsymbol{c}' \mid \exists \boldsymbol{c} \in \mathcal{C}, t_1 \leq t \leq t_2, \boldsymbol{c}' = traj(\boldsymbol{c}, t)\}.$$

The set of points reached at the end of an environment transition from $\mathcal{C}$ is denoted by $End_{\texttt{env}}(\mathcal{C}) := Pt_{[\delta,\delta]}(\mathcal{C})$.

Now to conform to our semantics, we carefully define the exact set of configurations reached right at the end of each *round* without transient configurations. A *frontier* set of configurations $End_{rnd}(\mathcal{C}, n)$ represents those configurations that are reached from $\mathcal{C}$ *exactly when* n rounds are completed. Formally,

$$End_{rnd}(\mathcal{C}, n) := \begin{cases} \mathcal{C}, \text{ if } n = 0 \\ End_{\texttt{env}}(End_{\texttt{prog}}(End_{rnd}(C, n-1))). \end{cases}$$

Finally, given a set of configurations $\mathcal{C}_0 \subseteq \mathbb{C}$, the set of all reachable states in n rounds is defined inductively:

$$Reach(\mathcal{C}_0, n) := \begin{cases} \mathcal{C}_0 \quad if n = 0 \\ Reach(\mathcal{C}_0, n-1) \qquad\qquad \text{otherwise} \\ \quad \cup\ Post(End_{rnd}(\mathcal{C}_0, n-1)) \\ \quad \cup\ Pt_{[0,\delta]}(End_{\texttt{prog}}(End_{rnd}(\mathcal{C}_0, n-1))), \end{cases}$$

Notice that all transient configurations during both program (computed by *Post*) and environment (computed by $Pt_{[0,\delta]}$) transitions are included in *Reach*.

4.2 Decomposing invariance verification

An *invariant* of a *Koord* program is a predicate that holds in all reachable configurations. Invariant requirements can express safety, for instance, that no two robots are ever too close (Collision avoidance), or that robots always stay within a designated area (Geofencing). Formally,

Definition 1. *An invariant inv is a predicate (Boolean valued function) over a configuration* $\boldsymbol{c}$ *such that, given a set of initial configurations of the system* $\mathcal{C}_0$*,*

$$\forall n \in \mathbb{N}, \forall \boldsymbol{c} \in Reach(\mathcal{C}_0, n), inv_{\boldsymbol{c}},$$

where $inv_{\boldsymbol{c}}$ *represents evaluating inv over* $\boldsymbol{c}$*.*

Definition 2. *A predicate inv is an* inductive invariant *of a system if given a set of initial configurations of the system* $\mathcal{C}_0$*, the following proof obligations (POs) hold:*

$$\forall \boldsymbol{c}_0 \in \mathcal{C}_0, inv_{\boldsymbol{c}_0} \tag{Base}$$

$$\forall \boldsymbol{c} \in \mathbb{C}, inv_{\boldsymbol{c}} \Rightarrow \forall \boldsymbol{c}' \in Reach(\{\boldsymbol{c}\}, 1), inv_{\boldsymbol{c}'} \tag{1}$$

[1] For different platforms, this function could be defined in closed form, as solutions of differential equations, or in terms of a numerical simulator.

That is, *inv* holds in the initial configuration(s) (PO (**??**)), and *inv* is preserved by both platform-independent discrete program transitions and the platform-dependent environment transitions (PO (**??**)). It is straightforward to prove that an inductive invariant is an invariant.

Our verification strategy for user-specified (inductive) invariants is to discharge the proof obligations. PO (**??**) is usually trivial. Therefore, we focus on PO (**??**). The *Koord* semantics enables us to *decouple* the environment and program transitions in *Reach*, and analyze each separately. PO (**??**) can be restated by expanding *Reach* and End_{rnd} as $\forall \boldsymbol{c} \in \mathbb{C}$,

$$inv_{\boldsymbol{c}} \Rightarrow \forall \boldsymbol{c}' \in Post(\{\boldsymbol{c}\}), inv_{\boldsymbol{c}'} \tag{2}$$

$$inv_{\boldsymbol{c}} \Rightarrow \forall \boldsymbol{c}' \in Pt_{[0,\delta]}(End_{\texttt{prog}}(\{\boldsymbol{c}\})), inv_{\boldsymbol{c}'}. \tag{3}$$

As in other concurrent systems, a major bottleneck in computing *Post* for PO (**??**) is the required enumeration of all $\boldsymbol{p} \in perms(\texttt{ID})$ permutations for all robots with reads/writes to shared contexts. We therefore seek for a stronger and easier to prove proof obligation using the lemma below:

Lemma 1. *Given any inductive predicate φ, for any configuration c satisfying φ, the following always holds*

$$\left(\bigwedge_{i \in ID} \bigwedge_{e \in Events} \forall \boldsymbol{c}' \in Post_e(\boldsymbol{c}, i), \varphi_{\boldsymbol{c}'}\right) \Rightarrow \forall \boldsymbol{c}' \in Post(\{\boldsymbol{c}\}), \varphi_{\boldsymbol{c}'}$$

The proof follows from expanding the definition of *Post* and inducting on each event sequence. As φ is preserved before and after every event transition $Post_e$ of every robot, the order of robot events do not violate φ. With Lemma **??**, we strengthen and rewrite PO (**??**) as $\forall \boldsymbol{c}, \boldsymbol{c}' \in \mathbb{C}$,

$$\bigwedge_{i \in \texttt{ID}} \bigwedge_{e \in Events} inv_{\boldsymbol{c}} \wedge \boldsymbol{c}' \in Post_e(\boldsymbol{c}, i) \Rightarrow inv_{\boldsymbol{c}'} \tag{4}$$

which no longer requires enumeration of all permutations. This is the main reason how our synchronous model of execution help verification scale.

We now discuss our approach to discharge PO (**??**). To further decouple program and environment transitions, we expand $Pt_{[0,\delta]}$ and rewrite PO (**??**) as $\forall \boldsymbol{c}, \boldsymbol{c}', \boldsymbol{c}'' \in \mathbb{C}$,

$$\begin{aligned}(&inv_{\boldsymbol{c}} \wedge \ \boldsymbol{c}' \in End_{\texttt{prog}}(\{\boldsymbol{c}\}) \\ &\quad \wedge \forall t \in [0,\delta], \boldsymbol{c}'' = traj(\boldsymbol{c}', t)) \Rightarrow inv_{\boldsymbol{c}''}.\end{aligned} \tag{5}$$

PO (**??**) requires reasoning about the dynamic behavior of *traj* during environment transitions, and it is a challenging research problem by itself. We introduce *port assumptions* to abstract away the continuous dynamic behavior.

Definition 3. *Given the traj function, a* port assumption $A(\cdot,\cdot)$ *is a predicate on* $\mathbb{C} \times \mathbb{C}$ *if* $\forall \boldsymbol{c}', \boldsymbol{c}'' \in \mathbb{C}$,

$$(\forall t \in [0,\delta], \boldsymbol{c}'' = traj(\boldsymbol{c}', t)) \Rightarrow A(\boldsymbol{c}', \boldsymbol{c}''). \tag{$PAsm$}$$

Port assumptions allow users to over-approximate *traj* and prove the invariant at hand. PO (**??**) can be validated with our *Koord* simulator or with other specialized tools for continuous dynamics (see Section **??**). Further, we know by definition $End_{\texttt{prog}}(\{\boldsymbol{c}\}) \subseteq Post(\{\boldsymbol{c}\})$, we can apply Lemma **??** in a similar way. Hence, with PO (**??**) and Lemma **??**, we can merge PO (**??**) and PO (**??**) and strengthen as $\forall \boldsymbol{c}, \boldsymbol{c}', \boldsymbol{c}'' \in \mathbb{C}$,

$$\bigwedge_{i \in \texttt{ID}} \bigwedge_{e \in Events} inv_{\boldsymbol{c}} \wedge \boldsymbol{c}' \in Post_e(\boldsymbol{c}, i) \wedge A(\boldsymbol{c}', \boldsymbol{c}'') \Rightarrow inv_{\boldsymbol{c}''} \tag{Ind}$$

where we can use our $\mathbb{K}$ symbolic execution semantics to construct the symbolic post configuration to represent $\boldsymbol{c}' \in Post_e(\boldsymbol{c}, i)$ for each event. Notice that PO (**??**) allows us to reason in per event fashion as well as per robot fashion.

4.3 Validating port assumptions: reachability analysis

Over three decades of research on verification of complex dynamical and hybrid systems [**?**,**?**] has led to the creation of a powerful toolbox for linear [**?**,**?**], nonlinear [**?**,**?**,**?**], and black-box systems [**?**]. Depending on the type and availability of the platform models, these tools can be used for discharging the port assumptions. Here, for the sake of completing the picture, we briefly mention how the traces from the *Koord* simulator together with the DryVR tool [**?**], could be used to check port assumptions for various platforms or find counterexamples.

DryVR uses numerical simulations to learn the sensitivity of the trajectories of the vehicle to changes in initial conditions, with a certain confidence level. Then it uses this sensitivity and additional simulations to either prove the given invariant (in our case the port assumption) or find a counter-example. Under certain robustness assumptions, this process is also guaranteed to terminate. We used *Koord* simulator to generate traces of a vehicle (and quadcopter) moving from a set of initial conditions to a target waypoint. From these traces, DryVR computes the reachable states (see Figure **??**). Notice that for the same relative distance between the initial position and the target, the quadcopter has a larger reachset than the vehicle because the former overshoots.

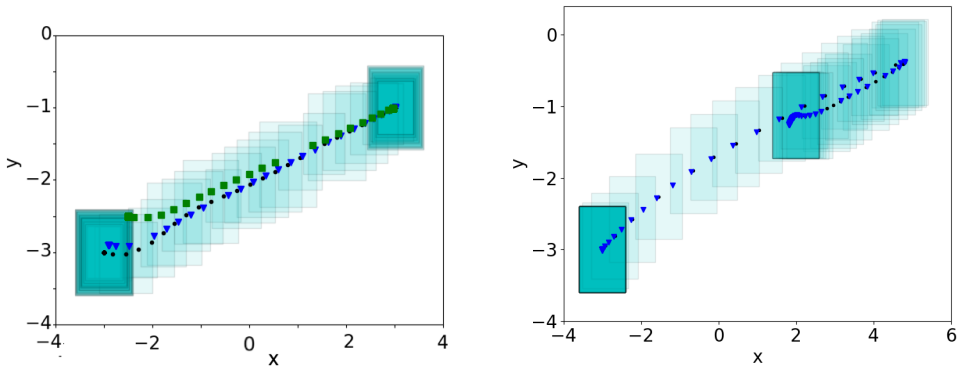

Fig. 9. *Left*: Reachsets for car. *Right*: Reachtube for drone

Following the notation in PO (**??**) and PO (**??**), variables and *primed copies* represents the variables in $\boldsymbol{c}$, $\boldsymbol{c}'$, and $\boldsymbol{c}''$ respectively.

Symbolically executing the event *TargetUpdate* (for robot i) generates the constraint:

$$E := \begin{pmatrix} \neg(i = \texttt{N\_SYS} - 1 \vee i = 0) \\ \wedge\ M.tgt'_i = (x[i-1] + x[i+1])/2 \wedge x'[i] = M.pos_i \\ \wedge\ \ldots \end{pmatrix}$$

Notice that it starts with the precondition and $M.tgt_i$ and $x[i]$ are updated according to the effect. We omit the rest of the formula that ensures the values of unmodified variables are unchanged such as $M.pos'_i = M.pos_i$ and $x'[j] = x[j]$ for $j \neq i$. Additionally, a more accurate version of Assumption **??** in Section **??** is

Port Assumption 2

$$A := M.pos''_i \in rect(M.pos'_i, M.tgt'_i) \wedge \ldots$$

where the rest of the formula ensuring unchanged values is omitted. Consequently, PO (**??**) becomes

Proof Obligation 1

$$(\forall t \in [0, \delta], M.pos''_i = f(M.pos'_i, M.tgt'_i, t)) \Rightarrow A$$

The invariant for the current configuration ($inv_{\boldsymbol{c}}$) is

$$I := M.pos_i \in rect(x_{min}, x_{max}) \wedge x[i] \in rect(x_{min}, x_{max})$$

and the invariant over primed configuration ($inv_{\boldsymbol{c}''}$) is

$$I'' := M.pos_i'' \in rect(x_{min}, x_{max}) \wedge x''[i] \in rect(x_{min}, x_{max})$$

Because there is only one event, PO (**??**) for LineForm then becomes

Proof Obligation 2 $\bigwedge_{i \in ID} I \wedge E \wedge A \Rightarrow I''$

Table **??** summarizes the verification of Proof Obligation **??** on systems with different N_SYS.

N_SYS	**dim**	T_K (s)	T_V (s)	**Safe**
3	1	4.90	9.09	✓
3	2	4.19	10.13	✓
4	1	4.79	12.21	✓
4	2	5.28	12.49	✓
4	3	5.06	12.77	✓
5	1	4.91	18.46	✓
5	2	5.60	18.91	✓
5	3	5.42	20.30	✓
10	1	10.92	32.34	✓
10	2	10.96	32.42	✓
10	3	11.34	33.61	✓
15	1	12.23	53.89	✓

Table 1. Summary of semantics based verification for LineForm. N is N_SYS, T_K is the symbolic post computation time in $\mathbb{K}$, T_V is the time taken for construction of constraints and verification in Z3. Robots moving along a line are represented by **dim** = 1, along a plane by **dim** = 2, and in a 3D space by **dim** = 3.

We can define a WI, a weaker invariant which simply constrains the position of each robot, and doesn't constrain the shared $x[i]$,

$$WI := M.pos_i \in rect[x_{min}, x_{max}]$$

Table **??** shows the results we obtained without the proof obligations as assumptions, and this weaker invariant on a system of robots moving in 2D. The verification procedure only tells us that the invariant is not inductive, it doesn't tell us whether the invariant doesn't hold, which is why we don't know whether the system is safe w.r.t the proposed invariant.

N_SYS	**constraint**	T_K (s)	T_V (s)	**Safe**
15	$E \wedge I \Rightarrow I'$	13.06	23.03	*Unknown*
15	$E \wedge WI \Rightarrow WI'$	9.92	18.26	*Unknown*
15	$E \wedge A \wedge WI \Rightarrow WI'$	11.24	32.64	*Unknown*

Table 2. Verification summary for weaker invariants on LineForm. The **constraint** column displays the induction hypothesis used in the verification.

5 Task Allocation and Mapping

5.1 : Distributed task allocation

in Figure **??** is a simple *Koord* program to solve distributed task allocation problem. The problem statement is as follows: Given a set of (possibly heterogeneous) robots, a safety distance $\epsilon > 0$, and a fixed sequence of points (tasks) $list = x_1, x_2, \ldots \in \mathbb{R}^3$, there are following two requirements: (a) every unvisited x_i in the sequence is *visited* exactly by one robot and (b) no two robots ever get closer than ϵ.

consists of two events (1) *Assign*, in which each robot looks for an unassigned task x from *list*; if there is a clear path to x then the robot assigns itself the task x, set the actuator port *Motion.path*, and shares

```
using Motion:
  actuators:
    List\<Point\> path
  sensors:
    Point position
    bool reached
    PathPlanner planner
local:
  bool on_task = s
  List\<Point\> curr_path
  Task cur_task

allread:
  List\<Point\> shared_paths[N_SYS]
allwrite:
  List\<Task\> all_tasks

Complete:
pre: on_task and Motion.reached
eff: on_task= False
shared_paths[pid]= [Motion.position]
```

```
Assign:
  pre: on_task
  eff:
    if len(all_tasks) == 0:
      stop
    else: atomic:
      for t in all_tasks:
        curr_path= Motion.planner(t.target)
        if pathIsClear(shared_paths, \
                       curr_path, pid):
          on_task= True
          cur_task= t
          break
      if on_task:
        all_tasks.remove(cur_task)
        shared_paths[pid]= curr_path
        Motion.path= curr_path
      else:
        shared_paths[pid]= [Motion.position]
```

Fig. 10. *Koord* code for robot i for the Distributed Task Allocation

its path with all other robots thru *shared_path*. Otherwise, it shares its position as the path. (2) *Complete*, which checks whether an robot has visited its assigned task. A path here is a list of points that a robot visits in sequence. The *Motion* module drives the robot along a path, as directed by the position value set at its actuator port *Motion.path*. The sensor port *Motion.planner* returns a path to the target of an unassigned task, and a (user-defined) function called *pathIsClear* is used to determine whether the currently planned path is within ϵ distance of any path in *shared_path*.

We only prove requirement (b) for . Table **??** summarizes the verification of these constraints with different number of robots.

Benchmark	N_SYS	T_K (s)	T_V (s)	**Safe**
Task	3	9.90	10.6	✓
Task	4	9.79	11.78	✓
Task	5	9.91	14.92	✓
Task	10	12.92	18.34	✓

Table 3. Summary of semantics based verification for . T_K is the symbolic post computation time in $\mathbb{K}$, T_V is the time taken for generation of constraints and verification in Z3 and N_SYS is the number of robots in the system.

5.2 Case Study: Mapping

For our last case study, we discuss the distributed grid mapping problem and our Mapping algorithm in *Koord* for the problem. The distributed grid mapping problem requires a set of robots to collaboratively mark the position of static *obstacles* within a given area D quantized by a *grid*, which any robot should avoid while moving in D. This problem is a simplified version of the distributed Simultaneous Localization and Mapping problem, a classical problem in robotics research. The difference comes with the assumption that the robots know their *global coordinates* within the area of deployment, and only attempt to map static

obstacles within this area. Further, the only sensors available for sensing obstacles are LIDAR based, and the robots are constrained to move in a 2D space.

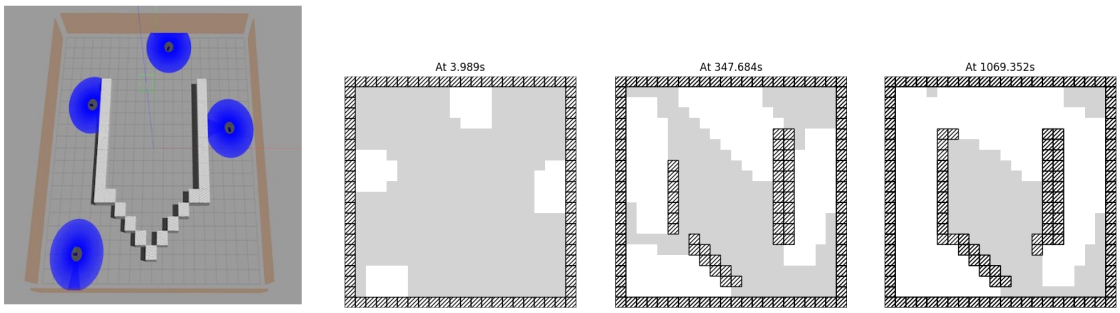

Fig. 11. Four cars in U-shape world in simulator (*Left*). Visualization of the global map at three different time stamps (*Right*)

Mapping algorithm works in the following manner. Each robot constructs a *local grid map* over D using sensors, and updates it using information from other robots shared via a *global grid map*. In Figure **??**, the *MotionWithScan* module provides a *pscan* sensor used to read the LIDAR scan of the actual robot. The other sensors and actuators *position, reached, planner, path* have the same functionality as that in the *Motion* module. The shared allwrite variable *map* is used to construct a shared map of obstacles within the domain D, and has type *GridMap*, which is a 2-D array representing a grid over D. The local variable *localMap* represents each robot's *local* knowledge of the domain D, and has the same type as D. There are three *events*: *NewPoint, LUpdate*, and *GUpdate*. A robot executing the *NewPoint* event, finds an unoccupied point to move to using a user defined function *pickFrontierPos* and plans a path to it using *MotionWithScan.planner*. It then updates its *localMap* from the shared variable *map*. The *LUpdate* event updates the *localMap* with scanned sensor data while the robot is in motion, and the *GUpdate* event updates the shared *map* with the updated *localMap* information corresponding to the scanned data.

A correctness requirement for Mapping is to ensure that, at any time, the global grid map *map* and all local maps $localMap_i$ should be consistent with the ground truth.

Table **??** summarizes the verification of these constraints on systems of different `N_SYS` .

Benchmark	**robots**(N)	T_K (s)	T_V (s)	**Safe**
DMap	3	9.23	14.53	✓
DMap	4	9.33	19.25	✓
DMap	5	9.19	24.30	✓
DMap	10	9.31	59.81	✓

Table 4. Summary of semantics based verification for DMap. T_K is the symbolic post computation time in $\mathbb{K}$, T_V is the time taken for generation of constraints and verification in Z3 and *robots*(N) is the number of robots in the system.

6 CyPhyHouse Architecture

A system running a application has three parts: an application program, a controller, and a plant. At runtime, the program executes within the runtime system of a single agent, or a collection of programs execute on different agents that communicate using shared variables. The plant consists of the hardware platforms of

```
using MotionWithScan
    sensors:
        Point position
        List\<Point, Scan\> pscan
        bool reached
        PathPlanner planner
    actuators:
        List\<Point\> path

allwrite:
    GridMap map

#omitting initialization
local:
   GridMap localMap
   Point target
   bool onpath = True
   List\<Grid\> obstacles

GUpdate:
  pre MotionWithScan.reached
  eff: atomic:
    map = merge(map, localMap)
    onpath = False

NewTarget:
  pre onpath
  eff:
    target = pickFrontierPos(map, \
                    MotionWithScan.position)
    obstacles = findObs(map)
    MotionWithScan.path = \
      MotionWithScan.planner(target, obstacles)
    if MotionWithScan.path = []:
      onpath = True
    else:
      onpath = False
    localMap = map

LUpdate:
  pre onpath and MotionWithScan.reached
  eff:
     for p, s in MotionWithScan.pscan:
        localMap = merge(localMap, \
                        scanToMap(p, s))
```

Fig. 12. *Koord* code for Distributed Mapping Application Mapping

the participating agents. The controller receives inputs from the program (through actuator ports), sends outputs back to the program (through sensor ports), and interfaces with the plant.

6.1 Compilation

The compiler included with generates Python code for the application using all the supported libraries, such as the implementation of distributed shared variables using message passing over WiFi, motion automata of the robots, high-level collision and obstacle avoidance strategies, etc. The application then runs with the Python *middleware* for . The compiler is written using Antlr (Antlr 4.7.2) in Java [**?**].[2] We use ROS to handle the low-level interfaces with hardware. To communicate between the high-level programs and low-level controllers, we use Rospy, a Python client library for ROS which enables the (Python) middleware to interface with ROS Topics and Services used for deployment or simulation.

6.2 Shared memory and Communication

At a high level, updates to a shared variable by one agent are propagated by the middleware, and become visible to other agents in the next round. The correctness of a program relies on agents having consistent values of shared variables. When an agent updates a shared variable, the middleware uses message passing to inform the other agents of the change. These changes should occur before the next round of computations.

 supports UDP based messaging over Wi-Fi for communication between robots to implement the shared memory. Any shared memory update translates to a update message which the agent broadcasts over WiFi.[3]

[2] Details of the grammar, AST, and IR design of the compiler are beyond the scope of this paper; however, description of the language and its full grammar are provided in [**?**] for the interested reader.

[3] The interested reader is referred to [**?**] for more details on the shared memory model, and its formal semantics.

The agents running a single distributed application are assumed to be running on a single network node, with little to no packet loss. However, the communication component of the middleware can be easily extended to support multi-hop networks as well.

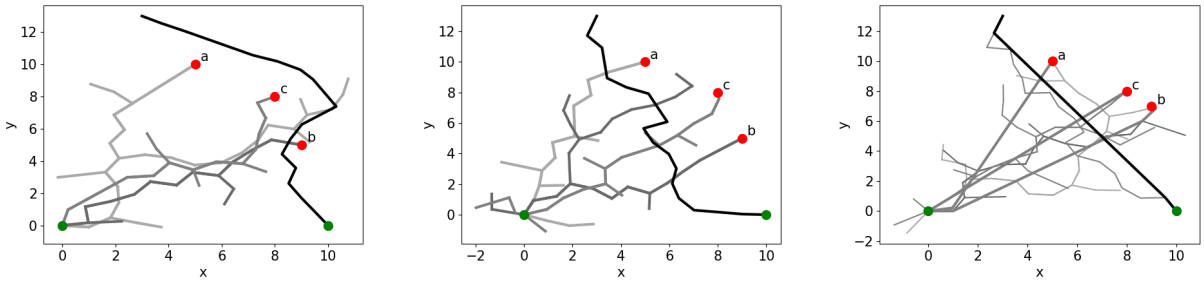

Fig. 13. Different planner can work with the same code. *Left* shows the xy plots of concurrently available paths during a round of the application using an RRT planner for two quadcopters. *Middle* shows the same configuration, where paths computed are not viable to be traversed concurrently. The green markers are current quadcopter positions, The black path is a fixed path, and the red points are unsassigned task locations. *Right* shows the same scenarios under which paths cannot be traversed concurrently, except that a different RRT-based planner (with path smoothing) is used.

6.3 Dynamics

If an application requires the agents to move, each agent uses an abstract class, *Motion automaton*, which must be implemented for each hardware model (either in deployment or simulation). This automaton subscribes to the required ROS Topics for positioning information of an agent, updates the *reached* flag of the motion module, and publishes to ROS topics for motion-related commands, such as waypoint or path following. It also provides the user the ability to use different path planning modules as long as they support the interface functions. Figure **??** shows two agents executing the *same application* using different path planners.

6.4 Portability

Apart from the dynamics, all aforementioned components of the middleware are platform-agnostic. Our implementation allows any agent or system simulating or deploying a program to use a configuration file (as shown in Figure **??**) to specify the system configuration, and the runtime modules for each agent, including the dynamics-related modules, while using the same application code.

7 Deployment Setup

7.1 Vehicles

As previously mentioned, the toolchain was developed with heterogeneous robotics platforms in mind. In order to demonstrate such capabilities, we have built both a car and a quadcopter.

Quadcopter The quadcopter was assembled from off-the-shelf hardware, with a 40cm × 40cm footprint. The main computing unit consists of a Raspberry Pi 3 B+ along with a Navio2 deck for sensing and motor control. Stabilization and reference tracking are handled by Ardupilot [**?**]. Between the middleware and Ardupilot we include a hardware abstraction layer to convert setpoint messages from the high-level language into MAVLINK using the mavROS library ([**?**]), so Ardupilot can parse them. Since the autopilot was originally meant to use a GPS module, we also convert the current quadcopter position into the Geographic Coordinate System before sending it to the controller.

Car Similar to the quadcopter, the car platform uses off-the-shelf hardware based on the open-source MIT RACECAR project [**?**]. The computing unit consists of an NVIDIA TX2 board. In the car platform, instead of using Ardupilot to handle the waypoint following, we wrote a custom ROS node uses the current position and desired waypoints to compute the input speed and steering angle using a Model Predictive Controller (MPC). The car has an electronic speed controller that handles low-level hardware control.

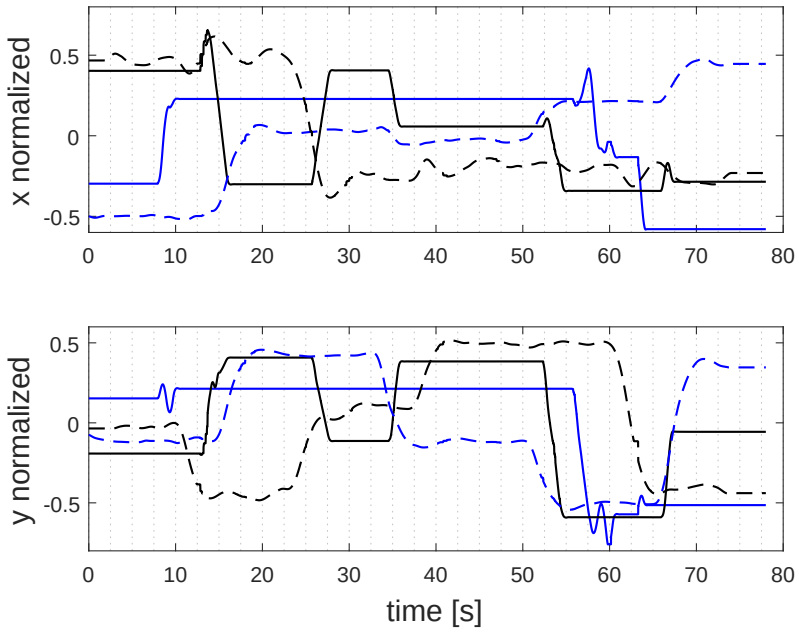

Fig. 14. *Top* shows the x *vs* t trajectories of the vehicles during an execution of the task, and *bottom* shows the y *vs* t trajectories. The vehicle positions were normalized to improve visualization. We can see concurrent movement when it is safe (for example at 13, 36, and 52 seconds), and only one robot moving or no robot moving when trying to compute a safe and collision-free path to a task.

7.2 Test arena and localization

We performed our experiments in a 7m × 8m × 3m arena equipped with 8 Vicon cameras. The Vicon system allows us to track the position of multiple robots with sub-millimeter accuracy, however, we note that the position data can come from any source (for example GPS, ultrawide-band, LIDAR), as long as all robots share the same coordinate system. While the motion capture system transmits all the data from a central computer, each vehicle only subscribes to its own position information. This was done to simplify experiments, as the goal of the paper is not to present new positioning systems. All coordination and de-conflicting across agents is performed based on position information shared explicitly through shared variables in the application.

7.3 Interface with middleware

As mentioned earlier in Section **??**, the same application can be deployed using different path planners, which are associated with the platform-specific *motion automaton* through interfaces defined by the middleware. Both vehicles use RRT-based path planners [**?**] to compute a path to the next task. For the car the RRT

uses a bicycle model to compute the feasible paths, while for the quadcopter the RRT assumes it can move in a straight line between points. The path generated is then forwarded to the robot via a ROS topic. The ROS topics required for positioning and setting waypoints of the vehicles were specified in the configuration. Each vehicle updates the *reached* topic when they reach a predefined ball around the destination. Note that the car has nonholonomic constraints, while the quadcopter has uncertain dynamics, so in other standard settings, a roboticist would have to develop a separate application for each platform.

7.4 Experiments with on upto four vehicles

The application of Section **??** was run in over 100 experiments with different combinations of cars and quadcopters. Figure **??** shows the (x, y)-trajectories of the vehicles in one specific trial run, in which two quadcopters and two cars were deployed. Careful examination of the figure shows that all the performance requirements of are achieved, with concurrent movement when different robots have clear paths to tasks, safe separation at all times, and agents getting blocked when there is no safe path found. In our experiments with up to 4 vehicles, we found that with fewer agents, there are fewer blocked paths, so each robot spends less time idling, but this non-blocking effect is superseded by the parallelism gains obtained from having multiple robots. For example, with three agents (2 quadcopters and 1 car, or 1 quadcopter and 2 cars) showing an average runtime of about 110 seconds for 20 tasks. The average runtime for the same with 4 agents across 70 runs was about 90 seconds. We experience zero failures, provided the wireless network conditions satisfied the assumptions stated in Section **??**.

8 CyphyHouse multi-robot simulator

We have built a high-fidelity simulator for testing distributed applications with large number of heterogeneous robots in different scenarios. Our middleware design allows us to separate the simulation of applications and communications from the physical models for different platforms. Consequently, the compiled applications together with the communication modules can run directly in the simulator—one instance for each participating robot, and only the physical dynamics and the robot sensors are replaced by their simulated counterparts. This flexibility enables users to test their applications under different scenarios and with various robot hardware platforms. Simpler physical models can be used for early debugging of algorithms; and the same code can be used later with more accurate physics and heterogeneous platforms. The simulator can be used to test different scenarios, with different numbers of (possibly heterogeneous) robots, with no modifications to the application code itself, rather simply modifying a configuration file as shown in Figure **??**. To our knowledge, this is the only simulator for distributed robotics providing such fidelity and flexibility.

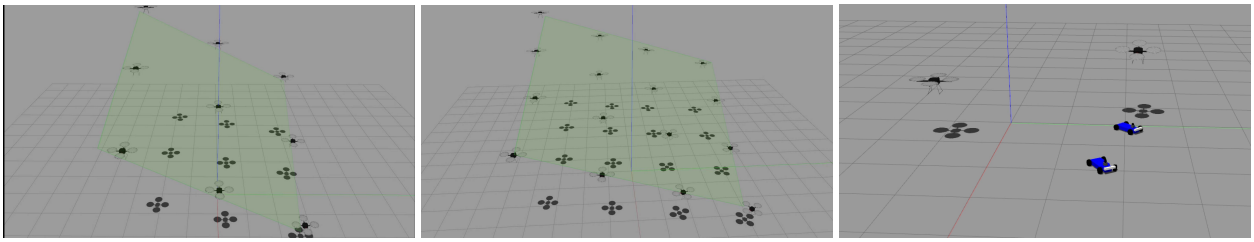

Fig. 15. simulator running different scenarios with the same application. *Left* shows simulation of 9 drones running application, *Middle* shows the application on 16 drones. Different scenarios are specified by changing the configuration file.*Right* shows a simulation of on heteterogenous robots.

8.1 Simulator Design

Simulating and communication To faithfully simulate the communication, our simulator spawns a process for each robot which encompasses all middleware threads. The communication handling threads in these

processes can then send messages to each other through broadcasts within the local network. To simulate robots on a single machine, we support specifying distinct network ports for robots in the configuration file. Since the communication is through actual network interfaces, our work can be extended to simulate different network conditions with existing tools in the future.

Physical Models and Simulated World Our simulated physical world is developed based on [**?**] and we provide a simulated positioning system to relay positions of simulated devices from to the middleware.

We integrate two robot models from the and ROS community, the car from the MIT RACECAR project [**?**] and the quadcopter from the hector quadrotor project [**?**]. Further, we implement a simplified version of position controller by modifying the provided default model. Users can choose between simplified models for faster simulation or original models for accuracy.

In addition to simulation, we also develop Gazebo plugins for visualization. Users may either use these to plot the movements or traces of the robots for real-time monitoring during experiments or visualize and analyze execution traces with Gazebo after experiments.

9 Remaining Goals and enhancements

Runtime monitoring and end-to-end verification

Partitioning into programmable multi-robot sub-systems CyPhyHouse provides distributed coordination across robots, wherein each robot is a node that performs individual computations potentially as part of a distributed task. Each robot also has individual sensing and actuation to determine its interaction with the environment. One of the new software enhancement goals includes implementation of a feature. This will allow partitioning of the overall distributed system into sub-components, where each sub-component potentially consists of multiple robots seen as a single unit. For instance, lightweight robots without individual computation units communicating with a base computer through ROS messages can be seen as one programmable unit in the overall distributed system. This adds a new level of abstraction to the Koord language itself, and opens up research questions about whether current verification techniques can be extended to these types of distributed systems, where each node may itself be a multi-robot system.

While an implementation is already possible with the currently existing CyPhyHouse middleware; the formal syntax and semantics for this language abstraction, and application of the verification approach through separation of platform dependent and independent components remains to be concretized.

Multi-application heterogeneous systems As another facet of the previous goal, I plan to explore the extension of the verification methodology to a a distributed system of robots running multiple applications , which communicate through shared variables *across applications* as well. For instance, the task and the mapping application can be combined to create an application in which robots perform a set of tasks in an unmapped grid. Preliminary proofs indicate that for this particular example it is a straightforward extension of the existing approach, and if realized, a user will be able to build complex applications incrementally, while ensuring the correctness of the application.